Algorithm Runtime Prediction: The State of the Art

Frank Hutter, Lin Xu, Holger H. Hoos, Kevin Leyton-Brown

Department of Computer Science
University of British Columbia
201-2366 Main Mall, BC V6T 1Z4, CANADA
{hutter, xulin730, hoos, kevinlb}@cs.ubc.ca

Abstract

Perhaps surprisingly, it is possible to predict how long an algorithm will take to run on previously unseen input data, using machine learning techniques to build a model of the algorithm's runtime as a function of domain-specific problem features. Such models have important applications to algorithm analysis, portfolio-based algorithm selection, and the automatic configuration of parameterized algorithms. Over the past decade, a wide variety of techniques have been studied for building such models. Here, we describe extensions and improvements of previous models, new families of models, and—perhaps most importantly—a much more thorough treatment of algorithm parameters as model inputs. We also describe novel features for predicting algorithm runtime for the propositional satisfiability (SAT), mixed integer programming (MIP), and travelling salesperson (TSP) problems. We evaluate these innovations through the largest empirical analysis of its kind, comparing to all previously proposed modeling techniques of which we are aware. Our experiments consider 11 algorithms and 35 instance distributions; they also span a very wide range of SAT, MIP, and TSP instances, with the least structured having been generated uniformly at random and the most structured having emerged from real industrial applications. Overall, we demonstrate that our new models yield substantially better runtime predictions than previous approaches in terms of their generalization to new problem instances, to new algorithms from a parameterized space, and to both simultaneously.

Keywords: Empirical hardness models, Response surface models, Performance prediction, Highly parametric algorithms, Propositional satisfiability, Mixed integer programming, Travelling salesperson problem
2010 MSC: 68T20

1. Introduction

NP-complete problems are ubiquitous in AI. Luckily, while these problems may be hard to solve on worst-case inputs, it is often feasible to solve even large problem instances that arise in practice. Less luckily, state-of-the-art algorithms often exhibit exponential runtime variation across instances from realistic distributions, even when problem size is held constant, and conversely the same instance can take exponentially

different amounts of time to solve depending on the algorithm used [29]. There is little theoretical understanding of what causes this variation, and it is nontrivial to determine how long a given algorithm will take to solve a given problem instance without incurring the potentially large cost of simply running the algorithm. Better answers to this question would advance our understanding of hardness beyond the worst case, and would also be practically useful in a variety of contexts.

Starting about a decade ago [68, 69], we have showed that machine learning methods can be used to build computationally inexpensive models that answer such questions with a high degree of accuracy, albeit imperfectly. These so-called *empirical hardness models* (EHMs) characterize problem instances by a set of cheaply computable features, and use supervised machine learning (specifically, regression) techniques to build models that predict algorithm runtime, based on a training set of previous algorithm runs.

EHMs have a wide range of applications. We initially conceived of them as a principled approach for gaining understanding of algorithm runtime beyond the worst-case analysis of problem hardness. In that vein, we developed techniques for identifying sets of features that correlate strongly with runtime. However, we quickly found that these models were also useful as engineering tools, in at least four ways [66, 67, 79, 46]. First, EHMs have been the key technology behind the first algorithm selection approaches achieving state-of-the-art performance for SAT. Specifically, the portfolio-based algorithm selector `SATzilla` [79, 108, 110] achieved excellent performance in the 2007 and the 2009 international SAT competitions, mainly based on the simple idea to predict the runtime of each algorithm in a portfolio and to then run the algorithm predicted to be best.[1] Second, they can be used to tune benchmark distributions for hardness [66, 69]. Third, EHMs can be used to automatically configure parametric algorithms, in at least two ways. The first is to model the runtime of a parameterized algorithm as dependent on both instance features and parameter values, and then, given a problem instance, to find a complete instantiation of parameter values (a so-called *parameter configuration*) with good predicted performance [46]. The second approach for automated algorithm configuration is to search for a single parameter configuration that achieves good aggregate performance on a given distribution of problem instances, by sequentially iterating between modeling steps and optimization steps that identify and evaluate promising configurations [53, 50, 51]. Fourth and finally, EHMs can serve as *surrogates* of algorithm performance, allowing algorithm runs to be cheaply simulated in cases where we are only interested in the algorithm's runtime. (This is the case, for example, in sophisticated algorithm configuration procedures like the one just outlined.)

Overall, we now find ourselves using statistical models more to improve algorithm performance than to understand problem hardness. Further, we have sometimes built models to predict objectives (such as the SAT competition scoring function [110] or the solution quality an optimization algorithm achieves in a fixed time [51]) that do not make much sense to describe as "hardness." Reflecting this broadened scope, here we use the term *empirical performance models* (EPMs), which we understand as an umbrella that includes EHMs.

[1] The most recent version of `SATzilla` employs cost-sensitive classification techniques instead of regression, yielding even better performance [111].

While the *idea* of modeling algorithm runtime is no longer new, we have made substantial recent progress in making these methods general, scalable and accurate. After a review of past work (Section 2) and the runtime prediction methods used by this work (Section 3), we describe four new contributions that jointly help us to advance and identify the current state of the art.

1. We introduce new, more sophisticated modeling techniques (based on random forests and approximate Gaussian processes) and methods for modeling runtime variation due to the setting of a large number of (categorical and continuous) algorithm parameter values (Section 4).
2. We present novel instance features for SAT, mixed integer programming (MIP) and the traveling salesperson problem (TSP)—in particular, novel probing features and timing features—yielding comprehensive sets of 138, 148, and 64 features for SAT, MIP, and TSP, respectively (Section 5).
3. To assess the impact of these advances and to determine the current state of the art, we performed what we believe is the most comprehensive evaluation of runtime prediction methods to date. Specifically, we evaluated all methods of which we are aware, based on performance data for 11 algorithms and 35 instance distributions spanning SAT, TSP and MIP and considering three different problems: predicting runtime on novel instances (Section 6), novel parameter configurations (Section 7), and both novel instances *and* configurations (Section 8).
4. Techniques from the statistical literature on survival analysis offer ways to better handle data from runs that were terminated prematurely. While these techniques were not used in most previous work—leading us to omit them from the comparison above—we show how to leverage them to achieve further improvements to our best-performing model, random forests (Section 9).[2]

2. An Overview of Related Work

Because the problems have been considered by substantially different communities, we separately consider related work on predicting the runtime of nonparametric and parametric algorithms.

2.1. Related Work on Predicting Runtime of Nonparametric Algorithms

To our knowledge, our own past work was the first to use statistical models to predict the runtime of algorithms for solving NP-complete problems. We began by studying the winner determination problem in combinatorial auctions [68, 69], and showed that accurate runtime predictions could be made for several different solvers and a wide variety of instance distributions. We considered a variety of different regression methods (including lasso regression, multivariate adaptive regression splines,

[2] We used early versions of the new modeling techniques described in Section 4, as well as the extensions to censored data described in Section 9 in recent conference and workshop publications on algorithm configuration [53, 50, 51, 49]. This article is the first to comprehensively evaluate the quality of these models.

and support vector machine regression), but in the end settled on a relatively simpler method: ridge regression with preprocessing to select an appropriate feature subset, a quadratic basis function expansion, and a log transformation of the response variable. (We formally define this and other regression methods in Section 3.) This work has found applications in solving the *algorithm selection* problem [84, 94] of deciding, on a per-instance basis, which of a finite set of algorithms should be run in order to optimize some performance objective, such as expected runtime. Focusing on SAT solving, we applied these models in our portfolio-based algorithm selection method `SATzilla`, devising various extensions along the way [79, 108, 110].

In a particularly noteworthy precursor to our work from the parallel computing literature in the mid-1990s, Brewer used linear regression models to predict the runtime of different implementations of portable, high-level libraries for multiprocessors, aiming to automatically select the best implementation on a novel architecture [16, 17]. Like us, Brewer used linear regression models to predict runtime based on empirical measurements of black-box algorithms. However, he focused on sub-quadratic-time algorithms, which do not exhibit the extreme runtime variability typical of algorithms for solving NP-complete problems. He thus used only a small set of basic instance features and a simpler and less flexible regression framework than we have found necessary.

Due to the exponential runtime variation often exhibited by algorithms for solving combinatorial problems, it is common practice to terminate unsuccessful runs after they exceed a so-called *captime*. Capped runs only yield a *lower bound* on algorithm runtime, but are typically treated as having succeeded at the captime. Gagliolo and co-authors [26, 25] were the first to apply techniques from the statistical literature on survival analysis to more appropriately handle this partially *right-censored* data in their work on dynamic algorithm portfolios. We used similar techniques for `SATzilla`'s runtime predictions [108] and in recent work on model-based algorithm configuration [49].

Recently, Smith-Miles and coauthors have published a series of papers on learning-based approaches for characterizing instance hardness for a wide variety of hard combinatorial problems [99, 98, 95, 97, 96, 100]. Their work has considered a range of tasks, including not only performance prediction, but also clustering, classification into easy/hard instances, and visualization. In the context of performance prediction, on which we focus in this article, their work only used neural network models and did not consider parametric algorithms. Also recently, Huang et al. [43] applied linear regression techniques to the modeling of algorithms with low-order polynomial runtimes. Haim & Walsh [35] extended linear methods to the problem of making online estimates about a SAT solver's runtime. A somewhat less related vein of research has not directly addressed runtime prediction, but has applied supervised classification to select the fastest algorithm for a problem instance [31, 27, 32, 28, 111] or to judge whether a particular run of a randomized algorithm would be good or bad [40].

Other research has aimed to identify single quantities that correlate with an algorithm's runtime. A famous early example is the clauses-to-variables ratio for uniform-random 3-SAT [18, 75]. Earlier still, Knuth showed how to use random probes of a search tree to estimate its size [63]; subsequent work refined the approach [72, 62]. We incorporated such predictors as features in our own work, and so do not evaluate them separately. (We do note, however, that we have found Knuth's tree-size estimate to be very useful for predicting runtime in some domains, such as complete SAT solvers on

unsatisfiable 3-SAT instances.) The literature on search space analysis has proposed a variety of quantities correlated with the runtimes of (mostly) local-search solvers. Prominent examples include fitness distance correlation [60] and autocorrelation length (ACL) [104]. With one exception (ACL for TSP) we have not included such measures in our feature sets, as computing them can be quite expensive.

2.2. Related Work on Predicting Runtime of Parametric Algorithms

In principle, it is not particularly harder to predict the runtimes of parametric algorithms than the runtimes of their nonparametric cousins: parameters can be treated as additional inputs to the model (notwithstanding the fact that they describe the algorithm rather than the problem instance, and hence are directly controllable by the experimenter), and a model can be learned in the standard way. In past work, we pursued precisely this approach, using both linear regression models and exact Gaussian processes to model the dependency of runtime on both instance features and algorithm parameter values [46]. However, this direct application of our methods designed for nonparametric algorithms is effective only for small numbers of continuous-valued parameters (*e.g.*, the experiments in [46] considered only two parameters). Different methods are more appropriate when an algorithm's parameter space becomes very large. In particular, a careful sampling strategy must be used, making it necessary to consider issues raised in the statistics literature on experimental design. Separately, models must be adjusted to deal with discrete and categorical parameters (*e.g.*, determining which of several possible heuristics to use; switching on or off a block of code).

The experimental design literature uses the term *response surface model (RSM)* to refer to a predictor that describes a process with controllable input parameters and that can generalize from observed data to new, unobserved parameter settings [see, *e.g.*, 13, 12]. The resulting function is called a *surrogate*, as it can be used to simulate the results of new experiments. The *design and analysis of computer experiments (DACE)* [86, 88] approach uses surrogates (usually in the form of Gaussian process models, also known as Kriging models) to model the outputs of computer programs such as climate simulations. Recently, we and others have leveraged models that predict an algorithm's runtime as a function of its parameter settings to find performance-optimizing settings using this DACE paradigm [see, *e.g.*, 7, 6, 52, 50].

Most of the literature on response surface models has limited its consideration to algorithms running on single problem instances and algorithms only with continuous input parameters. Beyond our own work, there are only a few papers that relax these assumptions: Bartz-Beielstein and Markon [8] support categorical algorithm parameters (using regression tree models), and two existing methods consider predictions across both different instances and parameter settings. First, Ridge & Kudenko [85] applied an analysis of variance (ANOVA) approach to detect important parameters, using linear and quadratic models. Second, Chiarandini & Goegebeur [19] noted that factors representing instance characteristics should be treated as random effects (as opposed to fixed effects, which are due to controllable inputs). Their resulting mixed-effects models are linear and, like ANOVA, assume Gaussian performance distributions. We note that this normality assumption is much more realistic in the context of predicting solution quality of local search algorithms (the problem addressed in [19]) than in the context of the algorithm runtime prediction problem we tackle here.

3. Methods Used in Related Work

We now formally define the different machine learning methods that have been used to predict algorithm runtimes: ridge regression (used by [16, 17, 68, 69, 79, 46, 108, 110, 43]), neural networks (see [98]), Gaussian process regression (see [46]), and regression trees (see [8]). This section provides the basis for the experimental evaluation of different methods in Sections 6, 7, and 8; thus, we also discuss some implementation details along the way.

3.1. Preliminaries

We describe a problem instance by a list of m features $\boldsymbol{z} = [z_1, \ldots, z_m]^\intercal$, drawn from a given *feature space* $\mathcal{F}$. These features must be computable by a piece of automated domain-specific code (usually provided by a domain expert) that efficiently extracts characteristics for any given problem instance (typically, in low-order polynomial time w.r.t. to the size of the given problem instance). We describe the *configuration space* of a parameterized algorithm with k parameters $\theta_1, \ldots, \theta_k$ with respective domains $\Theta_1, \ldots, \Theta_k$ by a subset of the cross-product of parameter domains: $\boldsymbol{\Theta} \subseteq \Theta_1 \times \cdots \times \Theta_k$. The elements of $\boldsymbol{\Theta}$ are complete instantiations of the algorithm's k parameters, and we refer to them as *configurations*. Taken together, the configuration and the feature spaces define the *input space*: $\mathcal{I} = \boldsymbol{\Theta} \times \mathcal{F}$.

Let $\Delta(\mathbb{R})$ denote the space of probability distributions over the real numbers; we will use these real numbers to represent an algorithm performance measure, such as runtime in seconds on some reference machine. (In principle, EPMs can predict any type of performance measure that can be evaluated in single algorithm runs, such as runtime, solution quality, memory usage, energy consumption, and communication overhead.) Given an algorithm $\mathcal{A}$ with configuration space $\boldsymbol{\Theta}$ and a distribution of instances with feature space $\mathcal{F}$, an EPM is a stochastic process $f : \mathcal{I} \mapsto \Delta(\mathbb{R})$ that defines a probability distribution over performance measures for each combination of a parameter configuration $\boldsymbol{\theta} \in \boldsymbol{\Theta}$ of $\mathcal{A}$ and a problem instance with features $\mathbf{z} \in \mathcal{F}$. The prediction of an entire distribution allows us to assess the model's *confidence* at a particular input, which is essential, *e.g.*, in model-based algorithm configuration [7, 6, 52, 50]. Nevertheless, since many of the methods we review yield only point-valued runtime predictions, our experimental analysis focuses on the accuracy of mean predicted runtimes. We study the accuracy of confidence values separately, and only for the models which define a proper probabilistic predictive distribution.

To construct an EPM for an algorithm $\mathcal{A}$ with configuration space $\boldsymbol{\Theta}$ on an instance set Π, we run $\mathcal{A}$ on various combinations of configurations $\boldsymbol{\theta}_i \in \boldsymbol{\Theta}$ and instances $\pi_i \in \Pi$, and record the resulting performance values y_i. We record the k-dimensional parameter configuration $\boldsymbol{\theta}_i$ and the m-dimensional feature vector $\boldsymbol{z}_i$ of the instance used in the ith run, and combine them to form a $p = k + m$-dimensional vector of *predictor variables* $\boldsymbol{x}_i = [\boldsymbol{\theta}_i^\intercal, \boldsymbol{z}_i^\intercal]^\intercal$. The training data for our regression models is then simply $(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)\}$. We use $\boldsymbol{X}$ to denote the $n \times p$ matrix containing $[\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n]^\intercal$ (the so-called *design matrix*) and $\boldsymbol{y}$ for the vector of performance values $[y_1, \ldots, y_n]^\intercal$.

We apply various transformations to this data to make it more amenable to modeling. In this article, we focus on runtime as a performance measure and use a log-

transformation, thus effectively predicting log-runtime.[3] In our experience, we have found this log-transformation to be very important due to the large variation in runtimes for hard combinatorial problems. We also transformed the predictor variables, discarding those input dimensions constant across all training data points and normalizing the remaining ones to have mean 0 and standard deviation 1 (*i.e.*, for each input dimension we subtracted the mean and then divided by the standard deviation).

For some instances, certain feature values were missing because of timeouts, crashes, or because they were undefined (because preprocessing already solved an instance). These *missing values* occur relatively rarely, so we use a simple mechanism for handling them. We disregard missing values for the purposes of normalization, and then set them to zero for training our models. This means that missing feature values are effectively assumed to be equal to the mean for the respective distribution and thus to be minimally informative. In some models (ridge regression and neural networks), this mechanism leads us to ignore missing features, since their weight is multiplied by zero.

3.2. Ridge Regression

Ridge regression [see, *e.g.*, 11] is a simple regression method that fits a linear function $f_{\boldsymbol{w}}(\boldsymbol{x})$ of its inputs $\boldsymbol{x}$. Due to its simplicity (both conceptual and computational) and its interpretability, combined with competitive predictive performance on most domains we studied, this is the method that we have used most frequently in the past for building EPMs [68, 69, 79, 46, 106].

Ridge regression works as follows. Let $\boldsymbol{X}$ and $\boldsymbol{y}$ be as defined above, let $\boldsymbol{I_p}$ be the $p \times p$ identity matrix, and let ϵ be a small constant. Then, compute the weight vector

$$\boldsymbol{w} = (\boldsymbol{X}^\intercal \cdot \boldsymbol{X} + \epsilon \cdot \boldsymbol{I_p})^{-1} \cdot \boldsymbol{X}^\top \cdot \boldsymbol{y}.$$

Given a new feature vector, $\boldsymbol{x}_{n+1}$, ridge regression predicts $f_{\boldsymbol{w}}(\boldsymbol{x}_{n+1}) = \boldsymbol{w}^\intercal \cdot \boldsymbol{x}_{n+1}$. Observe that with $\epsilon = 0$, we recover standard linear regression. The effect of $\epsilon > 0$ is to regularize the model by penalizing large coefficients $\boldsymbol{w}$; it is equivalent to a Gaussian prior favoring small coefficients under a Bayesian model (see, *e.g.*, [11]). A beneficial side effect of this regularization is that numerical stability improves in the common case where $\boldsymbol{X}$ is rank-deficient, or nearly so. The computational bottleneck in ridge regression with p input dimensions is the inversion of the $p \times p$ matrix $A = \boldsymbol{X}^\intercal \cdot \boldsymbol{X} + \epsilon \cdot \boldsymbol{I_p}$, which requires time cubic in p.

Algorithm runtime can often be better approximated by a polynomial function than by a linear one, and the same holds for log-runtimes. For that reason, it can make sense to perform a basis function expansion to create new features which are products of two or more original features. In light of the resulting increase in the number of features, a quadratic expansion is particularly appealing; indeed, we applied such an expansion in our earliest work on runtime prediction [68]. Formally, we augment each model input $\boldsymbol{x}_i = [x_{i,1}, \ldots, x_{i,p}]^\intercal$ with pairwise product inputs $x_{i,j} \cdot x_{i,l}$ for $j = 1, \ldots, p$ and $l = j, \ldots, p$.

[3] Due to the resolution of our CPU timer, runtimes below 0.01 seconds are measured as 0 seconds. To make $y_i = \log(r_i)$ well defined in these cases, we count them as 0.005 (which, in log space, has the same distance from 0.01 as the next bigger value measurable with our CPU timer, 0.02).

Even with ridge regularization, the generalization performance of linear regression (and, indeed, many other learning algorithms) can deteriorate when some inputs are uninformative or highly correlated with others; in our experience, it is difficult to construct sets of instance features that do not suffer from these problems. Instead, we reduce the set of input features by performing *feature selection*. Many different methods exist for feature expansion and selection, and we review three different ridge regression variants from the recent literature that only differ in these design decisions.

3.2.1. Ridge Regression Variant RR: Two-phase forward selection [108, 110]

For more than half a decade, we used a simple and scalable feature selection method based on forward selection [see *e.g.*, 34] to build the regression models used by `SATzilla` [108, 110]. This iterative method starts with an empty input set, greedily adds one linear input at a time, to minimize cross-validation error at each step, and stops when l linear inputs have been selected. It then performs a full quadratic expansion of these l linear features (using the original, un-normalized features, and then normalizing the resulting quadratic features again to have mean zero and standard deviation one). Finally, it carries out another forward selection with the expanded feature set, once more starting with an empty input set and stopping when q features have been selected. The reason for the two-phase approach is scalability: this method prevents us from ever having to construct a full quadratic expansion of our features. (For example, we often employ over 100 features and a million runtime measurements; thus, a full quadratic expansion of our features would require storing over 5 billion feature values.)

The computational complexity of forward selection can be reduced by exploiting the fact that the inverse matrix $(A')^{-1}$ resulting from including one additional feature can be computed incrementally by two rank-one updates of the previous inverse matrix A^{-1}, requiring quadratic time rather than cubic time [93].

In our experiments, we fixed the number of linear inputs to $l = 30$ in order to keep the result of a full quadratic basis function expansion manageable in size (with 1 million data points, the resulting matrix has $(\binom{30}{2} + 30) \cdot 1\,000\,000$, or about 500 million elements). The maximum number of quadratic terms q and the ridge penalizer ϵ are free parameters of this method; by default, we used $q = 20$ and $\epsilon = 10^{-3}$.

3.2.2. Ridge Regression Variant RR-elim: Eliminating redundant features [69]

In our earliest work we used a different feature selection method (see [69] for details). That method started with a quadratic feature expansion (similar to variant RR above) and then performed *backward selection*, iteratively elimininating, one at a time, the features whose value could be accurately predicted by a linear combination of the other features. This approach is infeasible if the number of features p is too large: the quadratic feature expansion yields $q = \binom{p}{2} + p \in \Theta(p^2)$ features, and the iterative backward selection requires time proportional to $q^5 = p^{10}$.[4] This approach would thus have been infeasible on our data, since we use substantially larger feature sets than in

[4]To see this, first note that eliminating the first feature requires building q models, each of which takes time cubic in q (rank-one updates do not apply since each of the models has a different response variable). Backward selection down to one feature then requires time proportional to $\sum_{i=1}^{q} i \cdot \Theta(i^3) \in \Theta(q^5)$.

Leyton-Brown et al. [69]. Nevertheless, it is one of the aims of this work to compare all previous approaches for building EPMs, and so we followed this method as faithfully as possible, given resource constraints. Specifically, we perform a quadratic feature expansion and then check for each resulting feature f_i whether it can be predicted by a linear combination of up to 10 other features (selected in turn by greedy forward selection), dropping f_i under the same conditions as in the original work: if the adjusted R^2 value of a linear model predicting f_i exceeds 0.99999. Also following the original work, we set the ridge penalizer to $\epsilon = 10^{-6}$.

3.2.3. Ridge Regression Variant SPORE-FoBa: Forward-backward selection [43]

Recently, Huang et al. [43] described a method for predicting algorithm runtime that they called Sparse POlynomial REgression (SPORE), which is based on ridge regression with forward-backward (FoBa) feature selection.[5] At the time, Huang et al. were unaware of our previous work and thus did not compare their SPORE-FoBa approach with our 2-phase forward selection. They did, however, conclude that SPORE-FoBa outperforms lasso regression, which is consistent with our own comparison to lasso (see, *e.g.*, [69]). In contrast to our two RR variants, SPORE-FoBa employs a cubic feature expansion (based on its own normalizations of the original predictor variables). Essentially, it performs a single pass of forward selection, at each step adding a small *set* of terms determined by a forward-backward phase on a feature's candidate set. Specifically, having already selected a set of terms T based on raw features S, SPORE-FoBa loops over all raw features $r \notin S$, constructing a candidate set T_r that consists of all polynomial expansions of $S \cup \{r\}$ that include r with non-zero degree and whose total degree is bounded by 3. For each such candidate set T_r, the forward-backward phase iteratively adds the best term $t \in T \setminus T_r$, if its reduction of root mean squared error (RMSE) exceeds a threshold γ (forward step), and then removes the worst term $t \in T$, if its reduction of RMSE is below $0.5 \cdot \gamma$ (backward step). This phase terminates when no single term $t \in T \setminus T_r$ can be added to reduce RMSE by more than γ. Finally, SPORE-FoBa's outer forward selection loop chooses the set of terms T resulting from the best of its forward-backward phases, and iterates until the number of terms in T reach a pre-specified maximum of t_{max} terms. SPORE-FoBa's free parameters are the ridge penalizer ϵ, t_{max}, and γ, with defaults $\epsilon = 10^{-3}$, $t_{\text{max}} = 10$, and $\gamma = 0.01$.

3.3. Neural Networks

Neural networks are a well-known regression method inspired by information processing in the human brain. The multilayer perceptron (MLP) is a particularly popular type of neural network that organizes single computational units ("neurons") in layers (input, hidden, and output layers), using the outputs of all units in a layer as the inputs of all units in the next layer. Each neuron n_i in the hidden and output layers with k inputs $\mathbf{a_i} = [a_{i,1}, \ldots, a_{i,k}]$ has an associated weight term vector $\mathbf{w_i} = [w_{i,1}, \ldots, w_{i,k}]$ and a bias term b_i, and computes a function $\mathbf{w_i}^\intercal \mathbf{a_i} + b_i$. For neurons in the hidden layer, the result of this function is further propagated through a nonlinear activation function

[5] Although not obvious from their publication [43], the authors confirmed to us that FoBa uses ridge rather than LASSO regression, and also gave us their original code.

$g : \mathbb{R} \to \mathbb{R}$ (which is often chosen to be $\tanh$). Given an input $\mathbf{x} = [x_1, \ldots, x_p]$, a network with a single hidden layer of h neurons $n_1, \ldots, n_h$ and a single output neuron n_{h+1} then computes output

$$\hat{f}(\mathbf{x}) = \left(\sum_{j=1}^{h} g(\mathbf{w_j}^\intercal \cdot \mathbf{x} + b_j) \cdot w_{h+1,j} \right) + b_{h+1}.$$

The $p \cdot h + h$ weight terms and $h + 1$ bias terms can be combined into a single weight vector $\mathbf{w}$, which can be set to minimize the network's prediction error using any continuous optimization algorithm (*e.g.*, the classic "backpropagation" algorithm performs gradient descent to minimize squared prediction error).

Smith-Miles & van Hemert [98] used an MLP with one hidden layer of 28 neurons to predict the runtime of local search algorithms for solving timetabling instances. They used the proprietary neural network software Neuroshell, but advised us to compare to an off-the-shelf Matlab implementation instead. We thus employed the popular Matlab neural network package NETLAB [76]. NETLAB uses activation function $g = \tanh$ and supports a regularizing prior to keep weights small, minimizing the error metric $\sum_i^N (\hat{f}(\mathbf{x_i}) - y_i)^2 + \alpha \cdot \mathbf{w}^\intercal \cdot \mathbf{w}$, where α is a parameter determining the strength of the prior. In our experiments, we used NETLAB's default optimizer (scaled conjugate gradients, SCG) to minimize this error metric, stopping the optimization after the default of 100 SCG steps. Free parameters are the regularization factor α and the number of hidden neurons h; we used NETLAB's default $\alpha = 0.01$ and, like Smith-Miles & van Hemert [98], $h = 28$.

3.4. Gaussian Process Regression

Stochastic Gaussian processes (GPs) [83] are a popular class of regression models with roots in geostatistics, where they are also called Kriging models [64]. GPs are the dominant modern approach for building response surface models [86, 59, 88, 6]. We believe that GPs have first been applied to algorithm runtime prediction in our own work [46], in which we found that GPs yielded better results than ridge regression but at a greater computational expense.

To construct a GP regression model, we first need to select a parameterized kernel function $k_\lambda : \mathcal{I} \times \mathcal{I} \mapsto \mathbb{R}^+$, characterizing the degree of similarity between two elements of the input space $\mathcal{I}$. We also need to determine the variance σ^2 of normally distributed observation noise, which in our setting corresponds to the variance of the target algorithm's runtime distribution. The predictive distribution of a zero-mean Gaussian stochastic process for response y_{n+1} at input $\boldsymbol{x}_{n+1}$ given training data $\mathcal{D} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_n, y_n)\}$, measurement noise variance σ^2, and kernel function k, is then the Gaussian

$$p(y_{n+1} \mid \boldsymbol{x}_{n+1}, \boldsymbol{x}_{1:n}, \boldsymbol{y}_{1:n}) = \mathcal{N}(y_{n+1} \mid \mu_{n+1}, \mathrm{Var}_{n+1}) \tag{1}$$

with mean and variance

$$\begin{aligned} \mu_{n+1} &= \boldsymbol{k_*}^\intercal \cdot [\boldsymbol{K} + \sigma^2 \cdot \boldsymbol{I_n}]^{-1} \cdot \boldsymbol{y}_{1:n} \\ \mathrm{Var}_{n+1} &= k_{**} - \boldsymbol{k_*}^\intercal \cdot [\boldsymbol{K} + \sigma^2 \cdot \mathbf{I}]^{-1} \cdot \boldsymbol{k_*}, \end{aligned}$$

where

$$\begin{aligned} \boldsymbol{K} &= \begin{pmatrix} k(\boldsymbol{x}_1, \boldsymbol{x}_1) & \dots & k(\boldsymbol{x}_1, \boldsymbol{x}_n) \\ & \ddots & \\ k(\boldsymbol{x}_n, \boldsymbol{x}_1) & \dots & k(\boldsymbol{x}_n, \boldsymbol{x}_n) \end{pmatrix} \\ \boldsymbol{k}_* &= (k(\boldsymbol{x}_1, \boldsymbol{x}_{n+1}), \dots, k(\boldsymbol{x}_n, \boldsymbol{x}_{n+1}))^\intercal \\ k_{**} &= k(\boldsymbol{x}_{n+1}, \boldsymbol{x}_{n+1}) + \sigma^2; \end{aligned}$$

see, *e.g.*, the book by Rasmussen & Williams [83] for a derivation. A variety of kernel functions are possible, but the most common choice for continuous inputs is the squared exponential kernel

$$K_{\text{cont}}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp\left(\sum_{l=1}^{p} \left(-\lambda_l \cdot (x_{i,l} - x_{j,l})^2\right)\right), \tag{2}$$

where $\lambda_1, \dots, \lambda_p$ are kernel parameters. This kernel is most appropriate if the response is expected to vary smoothly in the predictor variables $\boldsymbol{x}$.

The GP equations above assume fixed kernel parameters $\lambda_1, \dots, \lambda_p$ and fixed observation noise variance σ^2. These constitute the GP's *hyperparameters*, which are typically set by maximizing the *marginal likelihood* $p(\boldsymbol{y}_{1:n})$ of the data with a gradient-based optimizer; we refer to the book by Rasmussen & Williams [83] for details. The choice of optimizer can make a big difference in practice; we used the `minFunc` [90] implementation of a limited-memory version of BFGS [78].

Learning a GP model from data can be computationally expensive. Inverting the $n \times n$ matrix $[\boldsymbol{K} + \sigma^2 \cdot \mathbf{I_n}]$ takes $O(n^3)$ time and has to be done in every of the h hyperparameter optimization steps, yielding a total complexity of $O(h \cdot n^3)$. Subsequent predictions at a new input require only time $O(n)$ and $O(n^2)$ for the mean and the variance, respectively.

3.5. Regression Trees

Regression trees [15] are simple tree-based regression models. They are known to handle discrete inputs well; we believe that their first application to the prediction of algorithm performance, with the benefit that categorical parameters did not need to be encoded as real values, was by Bartz-Beielstein & Markon [8].

The leaf nodes of regression trees partition the input space into disjoint regions $R_1, \dots, R_M$, and use a simple model for prediction in each region R_m; the most common choice is to predict a constant c_m. This leads to the following prediction for an input point $\boldsymbol{x}$:

$$\hat{\mu}(\boldsymbol{x}) = \sum_{m=1}^{M} c_m \cdot \mathbb{I}_{\boldsymbol{x} \in R_m},$$

where the indicator function $\mathbb{I}_z$ takes value 1 if the proposition z is true and 0 otherwise. Note that since the regions R_m partition the input space, this sum will always involve exactly one non-zero term. We denote the subset of training data points in region R_m as $\mathcal{D}_m$. Under the standard squared error loss function $\sum_{i=1}^{n} \left(y_i - \hat{\mu}(\boldsymbol{x}_i)\right)^2$, the

error-minimizing choice of constant c_m in region R_m is then the sample mean of the data points in $\mathcal{D}_m$:

$$c_m = \frac{1}{|\mathcal{D}_m|} \sum_{\boldsymbol{x}_i \in R_m} y_i. \tag{3}$$

To construct a regression tree, we use the following standard recursive procedure, which starts at the root of the tree with all available training data points $\mathcal{D} = \{(\boldsymbol{x}_1, y_1), \ldots (\boldsymbol{x}_n, y_n)\}$. We consider binary partitionings of a given node's data along *split variables j* and *split points s*. For a real-valued split variable j, s is a scalar and data point $\boldsymbol{x}_i$ is assigned to region $R_1(j, s)$ if $x_{i,j} \leq s$ and to region $R_2(j, s)$ otherwise. For a categorical split variable j, s is a set, and data point $\boldsymbol{x}_i$ is assigned to region $R_1(j, s)$ if $x_{i,j} \in s$ and to region $R_2(j, s)$ otherwise. At each node, we select split variable j and split point s to minimize the sum of squared differences to the regions' means,

$$l(j, s) = \sum_{\boldsymbol{x}_i \in R_1(j,s)} (y_i - c_1)^2 + \sum_{\boldsymbol{x}_i \in R_2(j,s)} (y_i - c_2)^2, \tag{4}$$

where c_1 and c_2 are chosen according to Equation (3) as the sample means in regions $R_1(j, s)$ and $R_2(j, s)$, respectively. We continue this procedure recursively, finding the best split variable and split point, partitioning the data into two child nodes, and recursing into the child nodes. The process terminates when all training data points in a node share the same $\boldsymbol{x}$ values, meaning that no more splits are possible. This procedure tends to overfit the data, which can be mitigated by recursively pruning away branches that contribute little to the predictive accuracy of the model. We use cost-complexity pruning with 10-fold cross-validation to identify the best tradeoff between complexity and predictive quality (see the book by Hastie et al. [36] for details).

In order to predict the response value at a new input, $\boldsymbol{x}_i$, we *propagate $\boldsymbol{x}$ down the tree*, that is, at each node with split variable j and split point s, we continue to the left child node if $\boldsymbol{x}_{i,j} \leq s$ (for real-valued variable j) or $\boldsymbol{x}_{i,j} \in s$ (for categorical variable j), and to the right child node otherwise. The predictive mean for $\boldsymbol{x}_i$ is the constant c_m in the leaf that this process selects; there is no variance predictor.

3.5.1. Complexity of Constructing Regression Trees

If implemented efficiently, the computational cost for fitting a regression tree is small. At a single node with n data points of dimensionality p, it takes $O(p \cdot n \log n)$ time to identify the best combination of split variable and point, because for each continuous split variable j, we can sort the n values $\mathbf{x}_{1,j}, \ldots, \mathbf{x}_{n,j}$ and only consider up to $n - 1$ possible split points between different values. The procedure for categorical split variables has the same complexity: we consider each of the variable's k categorical values u_l, compute score $s_l = \text{mean}(\{y_i \mid \boldsymbol{x}_{i,j} = u_l\})$ across the node's data points, sort $(u_1, \ldots, u_k)$ by these scores, and only consider the k binary partitions with consecutive scores in each set. For the squared error loss function we use, the computation of $l(j, s)$ (see Equation (4)) can be performed in amortized time $O(1)$ for each of j's split points s, such that the total time required for determining the best split point of a single variable is $O(n \log n)$. The complexity of building a regression tree depends on how balanced it is. In the worst case, one data point is split off at a time, leading

to a tree of depth $n-1$ and a complexity of $O(p\sum_{i=1}^{n}(n-i)\log(n-i))$, which is $O(p\cdot n^2\log n)$. In the best case—a balanced tree—we have the recurrence relation $T(n)=v\cdot n\log n+2\cdot T(n/2)$, leading to a complexity of $O(p\cdot n\log^2 n)$. In our experience, trees are not perfectly balanced, but are much closer to the best case than to the worst case. For example, $10\,000$ data points typically led to tree depths between 25 and 30 (whereas $\log_2(10\,000)\approx 13.3$).

Prediction with regression trees is cheap; we merely need to propagate new query points $\mathbf{x}_{n+1}$ down the tree. At each node with continuous split variable j and split point s, we only need to compare $\mathbf{x}_{n+1,j}$ to s, an $O(1)$ operation. For categorical split variables, we can store a bit mask of the values in s to enable $O(1)$ member queries. In the worst case (where the tree has depth $n-1$), prediction thus takes $O(n)$ time, and in the best (balanced) case it takes $O(\log n)$ time.

4. New Modeling Techniques for EPMs

In this section we extend existing modeling techniques for EPMs, with the primary goal of improving runtime predictions for highly parametric algorithms. The methods described here draw on advanced machine learning techniques, but, to the best of our knowledge, our work is the first to have applied them for algorithm performance prediction. More specifically, we show how to extend all models to handle categorical inputs (required for predictions in partially categorical configuration spaces) and describe two new model families well-suited to modeling the performance of highly parameterized algorithms based on potentially large amounts of data: the projected process approximation to Gaussian processes and random forests of regression trees.

4.1. Handling Categorical Inputs

Empirical performance models have historically been limited to continuous-valued inputs; the only approach that has so far been used for performance predictions based on discrete-valued inputs is regression trees [8]. In this section, we first present a standard method for encoding categorical parameters as real-valued parameters, and then present a new kernel for handling categorical inputs more directly in Gaussian processes.

4.1.1. Extension of Existing Methods Using 1-in-$\mathcal{K}$ Encoding

A standard solution for extending arbitrary modeling techniques to handle categorical inputs is the so-called 1-in-$\mathcal{K}$ encoding scheme [see, *e.g.*, 11], which encodes categorical inputs with finite domain size $\mathcal{K}$ as $\mathcal{K}$ binary inputs. Specifically, if the ith column of the design matrix $\boldsymbol{X}$ is categorical with domain D_i, we replace it with $|D_i|$ binary indicator columns, where the new column corresponding to each $d\in D_i$ contains values $[\mathbb{I}_{x_{1,i}=d},\ldots,\mathbb{I}_{x_{n,i}=d}]^\intercal$, using the indicator function $\mathbb{I}_z$ (which is 1 iff z is true and 0 otherwise); for each data point, exactly one of the new columns is 1, and the rest are all 0. After this transformation, the new columns are treated exactly like the original real-valued columns, and arbitrary modeling techniques for numerical inputs become applicable.

4.1.2. A Weighted Hamming Distance Kernel for Categorical Inputs in GPs

We now define a new kernel for handling categorical inputs in Gaussian processes more natively than through a 1-in-$\mathcal{K}$ encoding. Our new kernel is similar to the standard squared exponential kernel of Equation (2), but instead of measuring the (weighted) squared distance, it computes a (weighted) Hamming distance:

$$K_{\text{cat}}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp\left(\sum_{l=1}^{p}(-\lambda_l \cdot \mathbb{I}_{x_{i,l} \neq x_{j,l}})\right). \tag{5}$$

For a combination of continuous and categorical input dimensions $\mathcal{P}_{\text{cont}}$ and $\mathcal{P}_{\text{cat}}$, we combine the two kernels:

$$K_{\text{mixed}}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \exp\left(\sum_{l \in \mathcal{P}_{\text{cont}}} \left(-\lambda_l \cdot (x_{i,l} - x_{j,l})^2\right) + \sum_{l \in \mathcal{P}_{\text{cat}}} \left(-\lambda_l \cdot \mathbb{I}_{x_{i,l} \neq x_{j,l}}\right)\right).$$

Although K_{mixed} is a straightforward adaptation of the standard kernel in Equation (2), we are not aware of any prior use of it. To use this kernel in GP regression, we have to show that it is positive definite:

Definition 1 (Positive definite kernel)**.** *A function $k : \mathcal{I} \times \mathcal{I} \mapsto \mathbb{R}$ is called a* positive definite kernel *iff it is*

- ***symmetric**: for any pair of inputs $\boldsymbol{x}_i, \boldsymbol{x}_j \in \mathcal{I}$, k satisfies $k(\boldsymbol{x}_i, \boldsymbol{x}_j) = k(\boldsymbol{x}_j, \boldsymbol{x}_i)$ and*
- ***positive definite**: for any n inputs $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \in \mathcal{I}$ and any n constants $c_1, \ldots, c_n \in \mathbb{R}$, k satisfies $\sum_{i=1}^{n}\sum_{j=1}^{n}(c_i \cdot c_j \cdot k(\boldsymbol{x}_i, \boldsymbol{x}_j)) \geq 0$.*

Proposition 2 (K_{mixed} is positive definite)**.** *For any combination of continuous and categorical input dimensions $\mathcal{P}_{cont}$ and $\mathcal{P}_{cat}$, K_{mixed} is a positive definite kernel function.*

Proof. We will use the facts that any constant is a positive definite kernel function, that the space of positive definite kernel functions is closed under addition and multiplication, and that k is a positive definite kernel function if there exists an embedding ϕ into some (potentially infinite-dimensional) space such that $k(x, z) = \phi(x)^\intercal \cdot \phi(z)$ [see, *e.g.*, 92]. First, consider a one-dimensional input with finite domain $\mathcal{I}$; we will begin by showing that K_{cat} is a positive definite kernel function for this domain. Let $a_1, \ldots, a_m$ denote the finitely many distinct elements of $\mathcal{I}$, and define an embedding ϕ into an m-dimensional space, mapping each element a_j to the m-dimensional indicator vector $\boldsymbol{v}_{a_j}$ that is zero everywhere except at position j, where it is one. Then define a kernel function $k_1(x, z)$ for $x, z \in \mathcal{I}$ as $k_1(x, z) = \phi(x)^\intercal \cdot \phi(z) = \boldsymbol{v}_x^\intercal \cdot \boldsymbol{v}_z = \sum_{j=1}^{m} \boldsymbol{v}_x(j) \cdot \boldsymbol{v}_z(j) = \mathbb{I}_{x=z}$. To bring this in the form of Equation (5), we add the constant kernel function $k_2(x, z) = c = \exp(-\lambda)/(1 - \exp(-\lambda))$, and then multiply by the constant kernel function $k_3(x, z) = 1/(1+c) = 1 - \exp(-\lambda)$. We can thus rewrite function K_{cat} as the product of positive definite kernels, thereby establishing that it, too, is positive definite:

$$\begin{aligned} K_{\text{cat}}(x, z) &= (k_1(x, z) + k_2(x, z)) \cdot k_3(x, z) \\ &= \begin{cases} 1 & \text{if } x = z \\ \exp(-\lambda) & \text{otherwise} \end{cases} \\ &= \exp(-\lambda \cdot \mathbb{I}(x \neq z)). \end{aligned}$$

It remains to show that K_{mixed} is positive definite. This follows immediately: K_{mixed} is a product of positive definite kernels (one K_{cat} kernel per categorical input and one K_{cont} kernel per continuous input), and the space of positive definite kernel functions is closed under multiplication. □

Kernel K_{mixed} has one hyperparameter λ_i for each input dimension, making it very expressive and prone to overfitting. Thus, we also experimented with two variations: (1) sharing the same hyperparameter λ across all input dimensions; and (2) sharing λ_1 across algorithm parameters and λ_2 across instance features. Despite the potential risk of overfitting, we found that K_{mixed} yielded the best results overall.

4.2. Scaling to Large Amounts of Data with Approximate Gaussian Processes

The time complexity of fitting Gaussian processes is cubic in the number of data points, which limits the amount of data that can practically be used to fit these models. To deal with this obstacle, the machine learning literature has proposed various approximations to Gaussian processes [see, *e.g.*, 82]. To the best of our knowledge, so far these approximate GPs have only been applied to runtime prediction in our work on parameter optimization [53] (considering parametric algorithms, but only single problem instances). We experimented with the Bayesian committee machine [102], the informative vector machine [65], and the projected process (PP) approximation [83]. All of these methods performed rather similarly, with a slight edge for the PP approximation. Below, we give the final equations for the PP's predictive mean and variance; for a derivation, see the book by Rasmussen & Williams [83].

The PP approximation to GPs uses a subset of a of the n training data points, the so-called *active set*. Let v be a vector consisting of the indices of these a data points. We extend the notation for exact GPs (see Section 3.4) as follows: let K_{aa} denote the a by a matrix with $K_{\text{aa}}(i,j) = k(\boldsymbol{x}_{v(i)}, \boldsymbol{x}_{v(j)})$ and let $\boldsymbol{K}_{\text{an}}$ denote the a by n matrix with $\boldsymbol{K}_{\text{an}}(i,j) = k(\boldsymbol{x}_{v(i)}, \boldsymbol{x}_j)$. The predictive distribution of the PP approximation is then a normal distribution with mean and variance

$$\begin{aligned} \mu_{n+1} &= \boldsymbol{k_*}^{\intercal} \cdot (\sigma^2 \cdot \boldsymbol{K}_{\text{aa}} + \boldsymbol{K}_{\text{an}} \cdot \boldsymbol{K}_{\text{an}}^{\intercal})^{-1} \cdot \boldsymbol{K}_{\text{an}} \cdot y_{1:n} \\ \text{Var}_{n+1} &= k_{**} - \boldsymbol{k_*}^{\intercal} \cdot \boldsymbol{K}_{\text{aa}}^{-1} \cdot \boldsymbol{k_*} + \sigma^2 \cdot \boldsymbol{k_*}^{\intercal} \cdot (\sigma^2 \cdot \boldsymbol{K}_{\text{aa}} + \boldsymbol{K}_{\text{an}} \cdot \boldsymbol{K}_{\text{an}}^{\intercal})^{-1} \cdot \boldsymbol{k_*}. \end{aligned}$$

We perform h steps of hyperparameter optimization based on a standard GP, trained using a set of a data points sampled uniformly at random without replacement from the n input data points. We then use the resulting hyperparameters and another independently sampled set of a data points (sampled in the same way) for the subsequent PP approximation. In both cases, if $a > n$, we only use n data points.

The complexity of the PP approximation is superlinear only in a; therefore, the approach is much faster when we choose $a \ll n$. The hyperparameter optimization based on a data points takes time $O(h \cdot a^3)$. In addition, there is a one-time cost of $O(a^2 \cdot n)$ for evaluating the PP equations. Thus, the time complexity for fitting the approximate GP model is $O([h \cdot a + n] \cdot a^2)$, as compared to $O(h \cdot n^3)$ for the exact GP model. The time complexity for predictions with this PP approximation is $O(a)$ for the mean and $O(a^2)$ for the variance of the predictive distribution [83], as compared to $O(n)$ and $O(n^2)$, respectively, for the exact version. In our experiments, we set $a = 300$ and $h = 50$ to achieve a good compromise between speed and predictive accuracy.

4.3. Random Forest Models

Random forests [14] are a flexible tool for regression and classification, and are particularly effective for high-dimensional and discrete input data. These characteristics make random forests an obvious choice for runtime predictions of highly parametric algorithms. Nevertheless, to the best of our knowledge, they have not yet been used for algorithm runtime predictions except in our own recent work on algorithm configuration [53, 50, 49, 51], which used a prototype implementation of the models we describe here. In the following, we describe the standard RF framework and some nonstandard implementation choices we made.

4.3.1. The Standard Random Forest Framework

A random forest (RF) consists of a set of regression trees. If grown to sufficient depths, regression trees are extraordinarily flexible predictors, able to capture very complex interactions and thus having low bias. However, this means they can also have high variance: small changes in the data can lead to a dramatically different tree. Random forests [14] reduce this variance by aggregating predictions across multiple different trees. (This is an alternative to the pruning procedure described previously; thus, the trees in random forests are not pruned, but are rather grown until each node contains no more than $n_{\min}$ data points.) These trees are made to be different by training them on different subsamples of the training data, and/or by permitting only a random subset of the variables as split variables at each node. We chose the latter option, using the full training set for each tree. (We did experiment with a combination of the two approaches, but found that it yielded slightly worse performance.)

Mean predictions for a new input $\boldsymbol{x}$ are trivial: predict the response for $\boldsymbol{x}$ with each tree and average the predictions. The predictive quality improves as the number of trees, B, grows, but computational cost also grows linearly in B. We used $B = 10$ throughout our experiments to keep computational costs low. Random forests have two additional hyperparameters: the percentage of variables to consider at each split point, *perc*, and the minimal number of data points required in a node to make it eligible to be split further, $n_{\min}$. We set $perc = 0.5$ and $n_{\min} = 5$ by default.

4.3.2. Modifications to Standard Random Forests

We introduce a simple, yet effective, method for quantifying predictive uncertainty in random forests. (Our method is similar in spirit to that of Meinshausen [74], who recently introduced quantile regression trees, which allow for predictions of quantiles of the predictive distribution; in contrast, we predict a mean and a variance.) In each leaf of each regression tree, in addition to the empirical mean of the training data associated with that leaf, we store the empirical variance of that data. To avoid making deterministic predictions for leaves with few data points, we round the stored variance up to at least the constant ${\sigma^2}_{\min}$; we set ${\sigma^2}_{\min} = 0.01$ throughout. For any input, each regression tree T_b thus yields a predictive mean μ_b and a predictive variance σ_b^2. To combine these estimates into a single estimate, we treat the forest as a mixture model of B different models. We denote the random variable for the prediction of tree T_b as L_b and the overall prediction as L, and then have $L = L_b$ if $Y = b$, where Y is a multinomial

variable with $p(Y = i) = 1/B$ for $i = 1, \ldots, B$. The mean and variance for L can then be expressed as follows:

$$\begin{aligned}
\mu = \mathbb{E}[L] &= \frac{1}{B}\sum_{b=1}^{B}\mu_b \\
\sigma^2 = \text{Var}(L) &= \mathbb{E}[\text{Var}(L|Y)] + \text{Var}(\mathbb{E}[L|Y]) \\
&= \left(\frac{1}{B}\sum_{b=1}^{B}\sigma_b^2\right) + \left(\mathbb{E}[\mathbb{E}(L|Y)^2] - \mathbb{E}[\mathbb{E}(L|Y)]^2\right) \\
&= \left(\frac{1}{B}\sum_{b=1}^{B}\sigma_b^2\right) + \left(\frac{1}{B}\sum_{b=1}^{B}\mu_b^2\right) - \mathbb{E}[L]^2 \\
&= \left(\frac{1}{B}\sum_{b=1}^{B}\sigma_b^2 + \mu_b^2\right) - \mu^2.
\end{aligned}$$

Thus, our predicted mean is simply the mean across the means predicted by the individual trees in the random forest. To compute the variance prediction, we used the law of total variance [see, *e.g.*, 105], which allows us to write the total variance as the variance across the means predicted by the individual trees (predictions are uncertain if the trees disagree), plus the average variance of each tree (predictions are uncertain if the predictions made by individual trees tend to be uncertain).

A second non-standard ingredient in our models concerns the choice of split points. Consider splits on a real-valued variable j. Note that when the loss in Equation (4) is minimized by choosing split point s between the values of $\boldsymbol{x}_{k,j}$ and $\boldsymbol{x}_{l,j}$, we are still free to choose the exact location of s anywhere in the interval $(\boldsymbol{x}_{k,j}, \boldsymbol{x}_{l,j})$. In typical implementations, s is chosen as the midpoint between $\boldsymbol{x}_{k,j}$ and $\boldsymbol{x}_{l,j}$. Instead, here we draw it uniformly at random from $(\boldsymbol{x}_{k,j}, \boldsymbol{x}_{l,j})$. In the limit of an infinite number of trees, this leads to a linear interpolation of the training data instead of a partition into regions of constant prediction. Furthermore, it causes variance estimates to vary smoothly and to grow with the distance from observed data points.

4.3.3. Complexity of Fitting Random Forests

The computational cost for fitting a random forest is relatively low. We need to fit B regression trees, each of which is somewhat easier to fit than a normal regression tree, since at each node we only consider $v = \max(1, \lfloor \mathit{perc} \cdot p \rfloor)$ out of the p possible split variables. Building B trees simply takes B times as long as building a single tree. Thus—by the same argument as for regression trees—the complexity of learning a random forest is $O(B \cdot v \cdot n^2 \cdot \log n)$ in the worst case (splitting off one data point at a time) and $O(B \cdot v \cdot n \cdot \log^2 n)$ in the best case (perfectly balanced trees). Our random forest implementation is based on a port of Matlab's regression tree code to C, achieving speedups of between one and two orders of magnitude.

Prediction with a random forest model entails predicting with B regression trees (plus an $O(B)$ computation to compute the mean and variance across those predictions). The time complexity of a single prediction is thus $O(B \cdot n)$ in the worst case and $O(B \cdot \log n)$ for perfectly balanced trees.

5. Domain-specific Instance Features

While the methods we have discussed so far could be used to model the performance of any algorithm for solving any problem, in our experiments, we investigated specific NP-complete problems. In particular, we considered the propositional satisfiability problem (SAT), mixed integer programming (MIP), and the travelling salesperson problem (TSP). SAT is the prototypical NP-hard decision problem and thus interesting from a theory perspective, but modern SAT solvers are also one of the most prominent approaches in hard- and software verification [81]. MIP is a canonical representation for constrained optimization problems with integer-valued and continuous variables. It serves as a unifying framework for NP-complete problems and combines the expressive power of integrality constraints with the efficiency of continuous optimization; as a consequence, it is very widely used both in academia and industry. Finally, TSP is one of the most widely studied NP-hard optimization problems, and also of considerable interest for industry [20].

The way we tailor EPMs to a particular domain is through the choice of instance features.[6] Here we describe comprehensive sets of features for SAT, MIP, and TSP. These include all features we are aware of from past work (nearly all from our own work, but also features due to Kadioglu et al. [61] and Smith-Miles et al. [99]), along with many new features that are introduced here for the first time. While all are polynomial-time computable, we note that some features can be computationally expensive for very large instances (*e.g.*, taking cubic time). In some applications such features will be reasonable (*e.g.*, in approximations of complex empirical analyses that take the features as a one-time input [48], and in domains with relatively small problem sizes); however, it may make sense to use only a subset of the features described here in runtime-sensitive applications.

Probing features are a family of features that deserves special mention. They are computed by briefly running an existing algorithm for the given problem on the given instance and extracting characteristics from the algorithm's trajectory—an idea closely related to that of landmarking in meta-learning [80]. Probing features can be defined with little effort for a wide variety of problems; indeed, in earlier work, we introduced the first probing features for SAT [79]. Here we introduce the first probing features for MIP and TSP. Another family of features we introduce here for the first time are *timing features*. These measure the time our other groups of features take to compute. Code and binaries for computing all our features, along with README files with additional details, are available online at http://www.cs.ubc.ca/labs/beta/Projects/EPMs/.

5.1. Features for Propositional Satisfiability (SAT)

Figure 1 summarizes 138 features for SAT. Since various preprocessing techniques are routinely used before applying a general-purpose SAT solver and typically lead to substantial reductions in instance size and difficulty (especially for industrial-like

[6] If features are unavailable for an NP-complete problem of interest, one alternative is to reduce the problem to SAT, MIP, or TSP—a polynomial-time operation—and then compute some of the features we describe here. We do not expect this approach to be computationally efficient, but observe that it extends the reach of existing EPM construction techniques to any NP-complete problem.

Problem Size Features:

1–2. **Number of variables and clauses in original formula**: denoted v and c, respectively

3–4. **Number of variables and clauses after simplification with SATElite**: denoted v' and c', respectively

5–6. **Reduction of variables and clauses by simplification**: (v-v')/v' and (c-c')/c'

7. **Ratio of variables to clauses**: v'/c'

Variable-Clause Graph Features:

8–12. **Variable node degree statistics:** mean, variation coefficient, min, max, and entropy

13–17. **Clause node degree statistics:** mean, variation coefficient, min, max, and entropy

Variable Graph Features:

18–21. **Node degree statistics:** mean, variation coefficient, min, and max

22–26. **Diameter:** mean, variation coefficient, min, max, and entropy

Clause Graph Features:

27–31. **Node degree statistics:** mean, variation coefficient, min, max, and entropy

32–36. **Clustering Coefficient:** mean, variation coefficient, min, max, and entropy

Balance Features:

37–41. **Ratio of positive to negative literals in each clause:** mean, variation coefficient, min, max, and entropy

42–46. **Ratio of positive to negative occurrences of each variable:** mean, variation coefficient, min, max, and entropy

47–49. **Fraction of unary, binary, and ternary clauses**

Proximity to Horn Formula:

50. **Fraction of Horn clauses**

51–55. **Number of occurrences in a Horn clause for each variable:** mean, variation coefficient, min, max, and entropy

DPLL Probing Features:

56–60. **Number of unit propagations:** computed at depths 1, 4, 16, 64 and 256

61–62. **Search space size estimate:** mean depth to contradiction, estimate of the log of number of nodes

LP-Based Features:

63–66. **Integer slack vector:** mean, variation coefficient, min, and max

67. **Ratio of integer vars in LP solution**

68. **Objective value of LP solution**

Local Search Probing Features, based on 2 seconds of running each of SAPS and GSAT:

69–78. **Number of steps to the best local minimum in a run:** mean, median, variation coefficient, 10th and 90th percentiles

79–82. **Average improvement to best in a run:** mean and coefficient of variation of improvement per step to best solution

83–86. **Fraction of improvement due to first local minimum:** mean and variation coefficient

87–90. **Coefficient of variation of the number of unsatisfied clauses in each local minimum:** mean and variation coefficient

Clause Learning Features (based on 2 seconds of running Zchaff_rand):

91–99. **Number of learned clauses:** mean, variation coefficient, min, max, 10%, 25%, 50%, 75%, and 90% quantiles

100–108. **Length of learned clauses:** mean, variation coefficient, min, max, 10%, 25%, 50%, 75%, and 90% quantiles

Survey Propagation Features

109–117. **Confidence of survey propagation:** For each variable, compute the higher of $P(true)/P(false)$ or $P(false)/P(true)$. Then compute statistics across variables: mean, variation coefficient, min, max, 10%, 25%, 50%, 75%, and 90% quantiles

118–126. **Unconstrained variables:** For each variable, compute $P(unconstrained)$. Then compute statistics across variables: mean, variation coefficient, min, max, 10%, 25%, 50%, 75%, and 90% quantiles

Timing Features

127–138. **CPU time required for feature computation:** one feature for each of 12 subsets of features (see text for details)

Figure 1: SAT instance features.

instances), we apply the preprocessing procedure SATElite [22] on all instances first, and then compute instance features on the preprocessed instances. The first 90 features, with the exception of features 22–26 and 32–36, were introduced in our previously published work [79]. They can be categorized as *problem size* features (1–7), *graph-based* features (8–36), *balance* features (37–49), *proximity to Horn formula* features (50–55), *DPLL probing* features (56–62), *LP-based* features (63–68), and *local search probing* features (69–90).

In our work on SATzilla [110, 109] over the last five years, we have introduced a variety of new features, which we describe here for the first time. Features 22–26 are based on the diameter of the variable graph [38]. For each node i in the variable graph, we compute the longest shortest path between i and any other node. As with most of the features that follow, we then compute various statistics over this vector (*e.g.*, mean, max); we do not state the exact statistics for each vector below but list them in Figure 1. Features 32–36 are based on the clustering coefficient of the clause graph, a measure of local graph cliqueiness. For each node in the clause graph, let p denote the number of edges present between the node and its neighbours, and let m denote the maximum possible number of such edges; we compute p/m for each node.

Our new *clause learning* features (91–108) are based on statistics gathered in 2-second runs of Zchaff_rand [73]. We measure the number of learned clauses (features 91–99) and the length of the learned clauses (features 100–108) after every 1000 search steps. Our *survey propagation* features (109–126) are based on estimates of variable bias in a SAT formula obtained using probabilistic inference [42]. We used VARSAT's implementation to estimate the probabilities that each variable is true in every satisfying assignment, false in every satisfying assignment, or unconstrained. Features 109–117 measure the confidence of survey propagation (that is, $\max(P_{\text{true}}(i)/P_{\text{false}}(i), P_{\text{false}}(i)/P_{\text{true}}(i))$ for each variable i) and features 118–126 are based on the $P_{\text{unconstrained}}$ vector.

Finally, our new *timing features* (127–138) measure the time taken by 12 different blocks of feature computation code: instance preprocessing by SATElite; problem size, variable-clause graph and balance features (1–17, 37–49); variable graph and proximity to Horn formula features (18–21, 50–55); diameter-based features (22–26); clause graph features (27–36); unit propagation features (56–60); search space size estimation (61–62); LP-based features (63–68); local search probing features (69–90) with SAPS and GSAT; clause learning features (91–108); and survey propagation features (109–126).

In our experiments, computing all of these features took 107 seconds on average and a maximum of 5 903 seconds; for 95% of instances, it took less than 200 seconds.

5.2. Features for Mixed Integer Programming (MIP)

Figure 2 summarizes 148 features for the mixed integer programming (MIP) problem. These include 101 features based on existing work [69, 44, 61], 42 new *probing* features, as well as 5 new *timing* features. Features 1–101 are primarily based on features for the combinatorial winner determination problem from our past work [69], generalized to MIP and so far only described in a Ph.D. thesis [44]. These features can be categorized as *problem type & size* features (1–25), *variable-constraint graph* features (26–49), *linear constraint matrix* features (50–73), *objective function* features (74–91), and *LP-based* features (92–95). We also integrated ideas from the feature set used by Kadioglu et al. [61] (namely *right-hand side* features (96–101), and the computation of separate

Problem Type:

1. **Problem type**: LP, MILP, FIXEDMILP, QP, MIQP, FIXEDMIQP, MIQP, QCP, or MIQCP, as attributed by CPLEX

Problem Size Features:

2–3. **Number of variables and constraints**: denoted n and m, respectively
4. **Number of non-zero entries in the linear constraint matrix, $\boldsymbol{A}$**
5–6. **Quadratic variables and constraints**: number of variables with quadratic constraints and number of quadratic constraints
7. **Number of non-zero entries in the quadratic constraint matrix, $\boldsymbol{Q}$**
8–12. **Number of variables of type**: Boolean, integer, continuous, semi-continuous, semi-integer
13–17. **Fraction of variables of type** (summing to 1): Boolean, integer, continuous, semi-continuous, semi-integer
18-19. **Number and fraction of non-continuous variables** (counting Boolean, integer, semi-continuous, and semi-integer variables)
20-21. **Number and fraction of unbounded non-continuous variables**: fraction of non-continuous variables that has infinite lower or upper bound
22-25. **Support size**: mean, median, vc, q90/10 for vector composed of the following values for bounded variables: domain size for binary/integer, 2 for semi-continuous, 1+domain size for semi-integer variables.

Variable-Constraint Graph Features: each feature is replicated three times, for $X \in \{C, NC, V\}$

26–37. **Variable node degree statistics**: characteristics of vector $(\sum_{c_j \in C} \mathbb{I}(A_{i,j} \neq 0))_{x_i \in X}$: mean, median, vc, q90/10
38–49. **Constraint node degree statistics**: characteristics of vector $(\sum_{x_i \in X} \mathbb{I}(A_{i,j} \neq 0))_{c_j \in C}$: mean, median, vc, q90/10

Linear Constraint Matrix Features: each feature is replicated three times, for $X \in \{C, NC, V\}$

50–55. **Variable coefficient statistics**: characteristics of vector $(\sum_{c_j \in C} A_{i,j})_{x_i \in X}$: mean, vc
56–61. **Constraint coefficient statistics**: characteristics of vector $(\sum_{x_i \in X} A_{i,j})_{c_j \in C}$: mean, vc
62–67. **Distribution of normalized constraint matrix entries, $\boldsymbol{A}_{i,j}/b_i$**: mean and vc (only of elements where $b_i \neq 0$)
68–73. **Variation coefficient of normalized absolute non-zero entries per row** (the normalization is by dividing by sum of the row's absolute values): mean, vc

Objective Function Features: each feature is replicated three times, for $X \in \{C, NC, V\}$

74-79. **Absolute objective function coefficients** $\{|c_i|\}_{i=1}^n$: mean and stddev
80-85. **Normalized absolute objective function coefficients** $\{|c_i|/n_i\}_{i=1}^n$, where n_i denotes the number of non-zero entries in column i of $\boldsymbol{A}$: mean and stddev
86-91. **squareroot-normalized absolute objective function coefficients** $\{|c_i|/\sqrt{n_i}\}_{i=1}^n$: mean and stddev

LP-Based Features:

92–94. **Integer slack vector:** mean, max, L_2 norm
95. **Objective function value of LP solution**

Right-hand Side Features:

96-97. **Right-hand side for $\leq$ constraints**: mean and stddev
98-99. **Right-hand side for $=$ constraints**: mean and stddev
100-101. **Right-hand side for $\geq$ constraints**: mean and stddev

Presolving Features:

102-103. **CPU times**: presolving and relaxation CPU time
104-107. **Presolving result features:** # of constraints, variables, non-zero entries in the constraint matrix, and clique table inequalities after presolving.

Probing Cut Usage Features:

108-112. **Number of specific cuts from CPLEX log**: clique cuts, Gomory fractional cuts, mixed integer rounding cuts, implied bound cuts, flow cuts
113-115. **Number of cuts from CPLEX API**: total clique cuts, total cover cuts, total cuts of any type

Probing Result features:

116-119. **Performance progress**: MIP gap achieved, # new incumbent found by primal heuristics, # of feasible solutions found, # of solutions or incumbents found
120-121. **Coverage**: # of nodes visited, # of iterations completed

Progress Over Time Features:

122-125. **Number of integer-infeasible variables at current node**: mean, median, vc, q90/10
126-127. **Nodes left to be explored**: mean, vc
128-131. **Gradient of objective function**: mean, median, vc, q90/10
132-135. **Gradient of best integer solution**: mean, median, vc, q90/10
136-139. **Gradient of achievable objective**: mean, median, vc, q90/10
140-143. **Gradient of MIP gap**: mean, median, vc, q90/10

Timing Features

144–148. **CPU time required for feature computation:** one feature for each of 5 groups of features (see text for details)

Figure 2: MIP instance features; for the variable-constraint graph, linear constraint matrix, and objective function features, each feature is computed with respect to three subsets of variables: continuous, C, non-continuous, NC, and all, V.

statistics for continuous variables, non-continuous variables, and their union). We extended existing features by adding richer statistics where applicable, namely medians, variation coefficients (vc), and percentile ratios (q90/q10) of vector-based features.

Our new set of MIP probing features is based on 5-second runs of CPLEX with default settings. These run statistics comprise 20 singleton features and 22 vector-based features, labeled as 102–121 and 122–143 in Figure 2, respectively. Since the CPLEX API does not provide rich enough information to track CPLEX's progress over time, we extracted most of these features from CPLEX's log file. The only exceptions were Features 113–115, which we did obtain through the API. The singleton features constitute 6 *presolving* features based on the output of CPLEX's presolving phase (102–107); 8 *probing cut usage* features describing the different cuts CPLEX used during probing (108–115); and 6 *probing result* features summarizing probing runs (116–121).

The remaining 22 *progress over time* features are based on vectors tracking the progress made by CPLEX by measuring the trajectory of six different quantities: the number of integer-infeasible variables, the number of nodes left to be explored, the objective value of the LP relaxation of the current node, the best integer feasible solution found so far, the best known dual bound, and the MIP gap. For each of these quantities, we parsed the corresponding entry from each line of CPLEX's log file output. For the number of integer-infeasible variables and the number of nodes left to be explored (122-127), the vector contains one element for each line. The remaining four quantities (128-143) measure improvement over time and their respective vectors thus contain the difference between two successive entries.

Finally, Features 144-148 capture the CPU time required for computing five different groups of features: variable-constraint graph, linear constraint matrix, and objective features for three subsets of variables ("continuous", "non-continuous", and "all", 26–91); LP-based features (92–95); and CPLEX probing features (102–143). The cost of computing the remaining features (1–25, 96–101) is small (linear in the number of variables or constraints).

Computing all 148 features took 8.0 seconds on average (0.8 seconds for Features 1–101; 7.2 seconds for the new probing features[7]), and a maximum of 53.6 seconds.

5.3. Features for the Travelling Salesperson Problem (TSP)

Figure 3 summarizes 64 features for the travelling salesperson problem (TSP). Features 1–50 are new, while Features 51–64 were introduced by Smith-Miles et al. [99]. The *problem size* feature (1) is the number of nodes in the given TSP. The *cost matrix* features (2–4) are statistics of the cost between two nodes. Our *minimum spanning tree* features (5–11) are based on constructing a minimum spanning tree over all nodes in the TSP: Features 5–8 are the statistics of the edge costs in the tree and Features 9–11 are based on its node degrees. Our *cluster distance* features (12–14) are based on the cluster distance between every pair of nodes, which is the minimum bottleneck cost of any path between them; here, the bottleneck cost of a path is defined as the largest cost along the path. Our *local search probing* features (15–32) are based on 20 short runs (1000 steps)

[7] We started CPLEX with a timeout of 5 seconds to compute probing features, but it did not always honour that timeout perfectly.

Problem Size Features:

1. **Number of nodes**: denoted n

Cost Matrix Features:

2–4. **Cost statistics:** mean, variation coefficient, skew

Minimum Spanning Tree Features:

5–8. **Cost statistics:** sum, mean, variation coefficient, skew

9–11. **Node degree statistics:** mean, variation coefficient, skew

Cluster Distance Features:

12–14. **Cluster distance:** mean, variation coefficient, skew

Local Search Probing Features:

15–17. **Tour cost from construction heuristic:** mean, variation coefficient, skew

18–20. **Local minimum tour length:** mean, variation coefficient, skew

21–23. **Improvement per step:** mean, variation coefficient, skew

24–26. **Steps to local minimum**: mean, variation coefficient, skew

27–29. **Distance between local minima**: mean, variation coefficient, skew

30–32. **Probability of edges in local minima**: mean, variation coefficient, skew

Branch and Cut Probing Features:

33–35. **Improvement per cut:** mean, variation coefficient, skew

36. **Ratio of upper bound and lower bound**

37–43. **Solution after probing:** Percentage of integer values and non-integer values in the final solution after probing. For non-integer values, we compute statics across nodes: min,max, 25%,50%, 75% quantiles

Ruggedness of Search Landscape:

44. **Autocorrelation coefficient**

Timing Features

45–50. **CPU time required for feature computation:** one feature each of 6 groups (see text)

Node Distribution Features (after instance normalization)

51. **Cost matrix standard deviation:** standard deviation of cost matrix after instance has been normalized to the rectangle $[(0,0),(400,400)]$.

52–55. **Fraction of distinct distances**: precision to 1, 2, 3, 4 decimal places

56–57. **Centroid:** the (x,y) coordinates of the instance centroid

58. **Radius:** the mean distances from each node to the centroid

59. **Area:** the are of the rectangle in which the nodes lie

60–61. **nNNd**: the standard deviation and coefficient variation of the normalized nearest neighbour distance

62–64. **Cluster:** #clusters / n , #outliers / n, variation of #nodes in clusters

Figure 3: TSP instance features.

of LK [71], using the implementation available from [21]. Features 15–17 are based on the tour length obtained by LK's Quick-Boruvka tour construction heuristic. Features 18–20, 21–23, and 24–26 are based on the tour length of local minima, the tour quality improvement per search step, and the number of search steps to reach a local minimum, respectively. Features 27–29 measure the Hamming distance between two local minima and features 30–32 describe the probability of edges appearing in any local minimum encountered during probing. Our *branch and cut probing* features (33–43) are based on 2-second runs of the `Concorde` solver. Features 33–35 measure the improvement of lower bound per cut, feature 36 is the ratio of upper and lower bound at the end of the probing run, and features 37–43 analyze the final LP solution. Feature 44 is the autocorrelation coefficient: a measure of the ruggedness of the search landscape, based on an uninformed random walk (see, *e.g.*, [39]). Finally, Features 45–50 measure the CPU time required for computing feature groups 2 to 7 (the cost of computing number of nodes can be ignored). Computing these 50 features took an average of 83.2 seconds, and at most 231 seconds.

Abbreviation	Reference Section	Description
RR	3.2	Ridge regression with 2-phase forward selection
RR-el	3.2	Ridge regression with backward elimination of redundant features
SP	3.2	SPORE-FoBa (ridge regression with forward-backward selection)
NN	3.3	Feed-forward neural network with one hidden layer
PP	4.2	Projected process (approximate Gaussian process); optimized via minFunc
RT	3.5	Regression tree with cost-complexity pruning
RF	4.3	Random forest

Table 1: Overview of our models.

Features 51–64 are due to Smith-Miles et al. [99], and we computed them using code obtained from the authors (available at http://www.vanhemert.co.uk/files/TSP-feature-extract-20120212.tar.gz). Features 51–61 are based on the spatial distribution of the nodes, and Features 62–64 are based on node clustering using the GDBSCAN algorithm [87]. A detailed description of these features can be found in [99].

6. Performance Predictions for New Instances

We now describe our evaluation of the modeling strategies discussed in Sections 3 and 4; Table 1 provides an overview of the models we evaluated. We begin by examining the accuracy achieved by each method for predicting the runtime of a variety of solvers for SAT, MIP, and TSP in their default configuration, the (only) problem considered by most past work. (We go on to consider predictive accuracy across different parameter configurations in Sections 7 and 8.) For brevity, we only present representative empirical results in this article. We provide the full results of our experiments in an online appendix at http://www.cs.ubc.ca/labs/beta/Projects/EPMs. All of our data and source code for replicating our experiments is available from the same site. We begin each experimental section with a description of the instances, solvers, and other setup used for those experiments.

6.1. Instances and Solvers

For SAT, we used a wide range of instance distributions (detailed in Appendix A.1). Briefly, `INDU`, `HAND`, and `RAND` are collections of industrial, handmade, and random instances from the international SAT competitions and races, and `COMPETITON` is their union. `SWV` and `IBM` are sets of software and hardware verification instances, and `SWV-IBM` is their union. Finally, `RANDSAT` is a subset of `RAND` containing only satisfiable instances. For all distributions except `RANDSAT`, we ran the popular tree search solver, `Minisat 2.0` [23]. For `INDU`, `SWV` and `IBM`, we also ran two more solvers: `CryptoMinisat` [101] (which won the SAT Race 2010, as well as a gold and a silver medal in the 2011 SAT competition) and `SPEAR` [5] (which has shown state-of-the-art performance on `IBM` and `SWV` with optimized parameter settings [45]). Finally, to evaluate predictions for local search algorithms, we used the `RANDSAT` instances, and considered two solvers: `tnm` [103] (which won the random satisfiable category of the 2009 SAT Competition) and the dynamic local search algorithm `SAPS` [54] (which we see as a baseline).

For MIP, we used two instance distributions from computational sustainability (`RCW` and `CORLAT`), one from winner determination in combinatorial auctions (`REG`), two unions

of these (CR := CORLAT ∪ RCW and CRR := CORLAT ∪ REG ∪ RCW), and a large and diverse set of publicly available MIP instances (BIGMIX). Details about these distributions are given in Appendix A.2. We used the two state-of-the-art commercial solvers CPLEX [56] and Gurobi [33] (versions 12.1 and 2.0, respectively) and the two strongest non-commercial solvers, SCIP [10] and lp_solve [9] (versions 1.2.1.4 and 5.5, respectively).

For TSP, we used three instance distributions (detailed in Appendix A.3): random uniform Euclidean instances (RUE), random clustered Euclidean instances (RCE), and TSPLIB, a heterogeneous set of prominent TSP instances. On these instance sets, we ran the state-of-the-art systematic and local search algorithms, Concorde [2] and LK-H [37]. For the latter, we computed runtimes as the time required to find an optimal solution.

6.2. Experimental Setup

To collect the algorithm runtime data used in this section, for each algorithm–distribution pair, we executed the algorithm on all instances of the distribution, measured its runtimes, and collected the results in a database. All algorithm runs used default parameters and were executed on a cluster of 55 dual 3.2GHz Intel Xeon PCs with 2MB cache and 2GB RAM, running OpenSuSE Linux 11.1; runtimes were measured as CPU time on these reference machines. We cut off each algorithm run after one CPU hour; this gives rise to *capped* runtime observations, because for each run that is terminated in this fashion, we only observe a lower bound on the runtime. Like most past work on runtime modeling, we simply counted such capped runs as having taken one hour. (In Section 9 we investigate alternatives and conclude that a better treatment of capped runtime data improves predictive performance for our best-performing model.)

We evaluated different model families by building models on a subset of the data and assessing their performance on data that had not been used to train the models. This can be done visually (as, *e.g.*, in the scatter plots in Figure 4 on Page 26), or quantitatively. We considered three complementary quantitative metrics to evaluate mean predictions $\mu_1, \ldots, \mu_n$ and predictive variances $\sigma_1^2, \ldots, \sigma_n^2$ given true performance values $y_1, \ldots, y_n$. *Root mean squared error (RMSE)* is defined as $\sqrt{1/n \sum_{i=1}^{n} (y_i - \mu_i)^2}$; *Pearson's correlation coefficient (CC)* is defined as $(\sum_{i=1}^{n} (\mu_i y_i) - n \cdot \bar{\mu} \cdot \bar{y}) / ((n-1) \cdot s_\mu \cdot s_y)$, where $\bar{x}$ and s_x denote sample mean and standard deviation of x; and *log likelihood (LL)* is defined as $\sum_{i=1}^{n} \log \varphi(\frac{y_i - \mu_i}{\sigma_i})$, where φ denotes the probability density function (PDF) of a standard normal distribution. Intuitively, LL is the log probability of observing the true values y_i under the predicted distributions $\mathcal{N}(\mu_i, \sigma_i^2)$. For CC and LL, higher values are better, while for RMSE lower values are better.

We used k-fold cross-validation (with $k = 10$ everywhere except in the first experiment, in which we used $k = 2$ to limit computational cost), and we report means of these measures across the k folds. Scatter plots show cross-validated predictions for a random subset of up to 1 000 data points.

6.3. Predictive Quality

Table 2 provides quantitative results for all benchmarks, and Figure 4 visualizes results. At the broadest level, we can conclude that most of the methods were able to capture enough about algorithm performance on training data to make meaningful predictions on test data, most of the time: easy instances tended to be predicted as

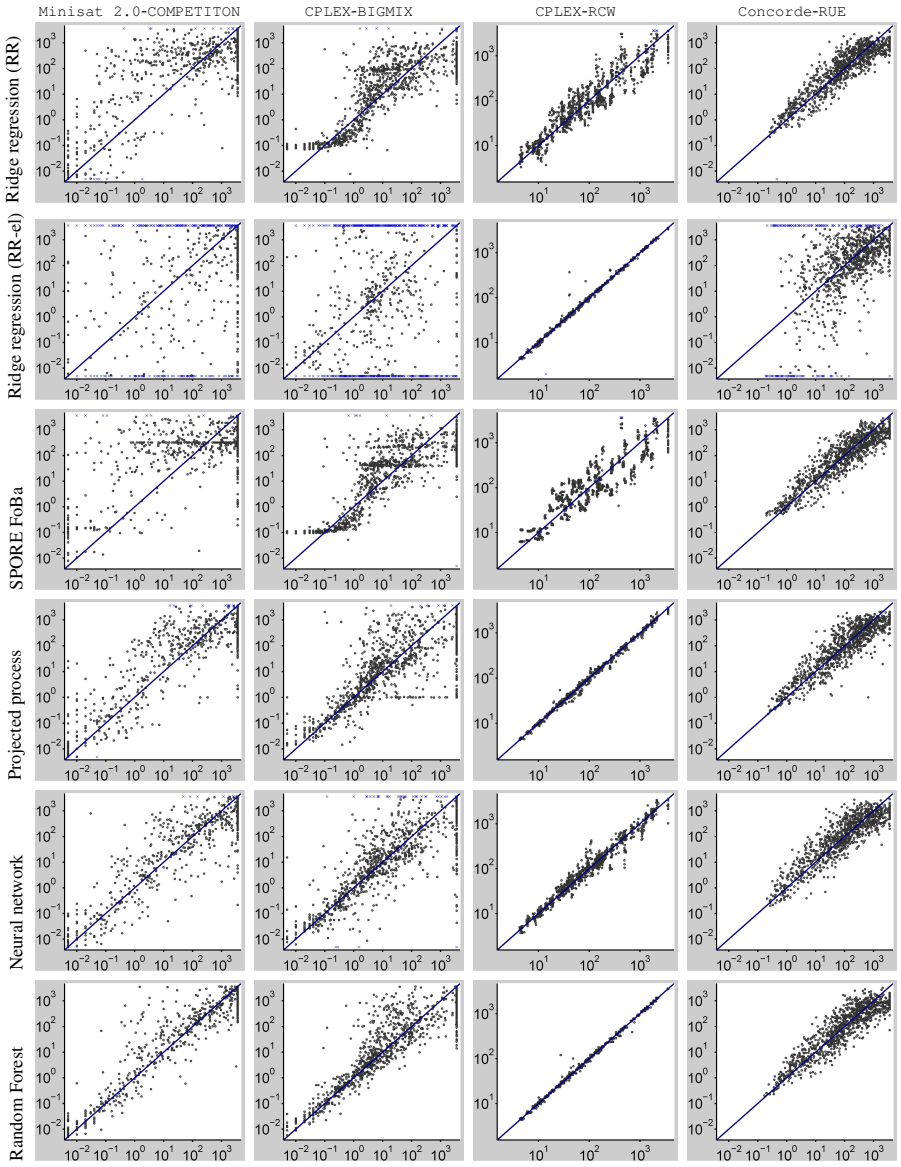


Figure 4: Visual comparison of models for runtime predictions on previously unseen test instances. The data sets used in each column are shown at the top. The x-axis of each scatter plot denotes true runtime and the y-axis cross-validated runtime as predicted by the respective model; each dot represents one instance. Predictions above 3 000 or below 0.001 are denoted by a blue cross rather than a black dot. Figures B.1–B.10 (in the online appendix) show equivalent plots for the other benchmarks and also include regression trees (whose predictions were similar to those of random forests but had larger spread).

	RMSE							Time to learn model (s)						
Domain	RR	RR-el	SP	NN	PP	RT	RF	RR	RR-el	SP	NN	PP	RT	RF
Minisat 2.0-COMPETITON	1.1	2.9E2	1.3	0.7	0.83	0.75	**0.55**	**4.9**	2.9E5	15	12	46	12	11
Minisat 2.0-HAND	1.2	3.2	1.3	0.77	0.87	0.92	**0.67**	3.7	2.6E5	8.4	3.8	49	3.5	**2.9**
Minisat 2.0-RAND	0.64	3.4	0.77	**0.41**	0.58	0.57	**0.41**	**4**	1.3E5	5.8	6.2	47	4.1	4.3
Minisat 2.0-INDU	0.97	86	1.0	0.99	0.86	0.93	**0.69**	3.6	2.3E5	7	3.2	44	3.8	**2.4**
Minisat 2.0-SWV-IBM	0.54	6.8	0.75	0.4	0.61	0.29	**0.21**	3.4	2E5	7.3	2.6	45	2.9	**1.5**
Minisat 2.0-IBM	0.63	76	12	0.38	0.45	0.38	**0.25**	3.4	2E5	4.5	1.6	43	1.6	**0.85**
Minisat 2.0-SWV	0.48	0.54	0.22	0.18	0.11	0.12	**0.1**	3.5	1.8E5	6.2	1.4	62	1.5	**0.62**
CryptoMinisat-INDU	0.97	20	1.0	1.2	1.0	1.0	**0.82**	3.5	2.3E5	6.8	3.2	44	2.9	**2.1**
CryptoMinisat-SWV-IBM	0.9	7.7	0.93	0.76	0.89	0.69	**0.55**	3.6	2E5	5.1	2.8	44	2.8	**1.5**
CryptoMinisat-IBM	0.74	1.4E2	0.95	0.63	0.64	0.6	**0.49**	3.4	1.9E5	5.5	1.6	43	1.6	**0.85**
CryptoMinisat-SWV	0.94	2.4	1.1	0.74	0.68	0.7	**0.57**	3.4	1.8E5	4.4	1.3	63	1.5	**0.62**
SPEAR-INDU	1.0	47	1.0	1.0	0.89	0.87	**0.73**	3.6	2.3E5	6.4	3.1	47	3.2	**2.3**
SPEAR-SWV-IBM	0.7	9.4	0.83	0.58	0.82	0.58	**0.44**	3.5	2E5	7.9	2.6	49	2.8	**1.5**
SPEAR-IBM	0.79	4.3E2	3.6	0.6	0.71	0.52	**0.45**	3.4	1.9E5	5.4	1.6	48	1.6	**0.89**
SPEAR-SWV	0.49	3.0	0.68	0.52	0.5	0.58	**0.44**	3.4	1.8E5	6.4	1.4	52	1.5	**0.64**
tnm-RANDSAT	1.0	8.6	1.1	1.1	0.95	1.2	**0.92**	3.7	1.2E5	7.8	3.8	49	4.3	**2.8**
SAPS-RANDSAT	0.97	4.9	1.1	0.83	0.77	1.0	**0.7**	3.6	1.2E5	7	3.8	44	3.8	**2.6**
CPLEX-BIGMIX	9E11	1.4E18	1.1E7	1.1	1.1	0.99	**0.72**	3.3	1.4E5	8.5	2.7	38	3.1	**1.9**
Gurobi-BIGMIX	3.1E12	3.5E19	6.2E2	1.8	1.3	1.5	**1.2**	3.2	1.4E5	3.9	2.7	38	3.2	**2**
SCIP-BIGMIX	2.6	1.9E19	1.3	0.99	0.99	0.76	**0.63**	3.4	1.4E5	6.2	2.7	37	2.9	**2.1**
lp_solve-BIGMIX	3.6	7.7E18	1.6	0.93	1.1	0.69	**0.57**	3.3	1.3E5	4	2.7	44	**1.6**	2.5
CPLEX-CORLAT	0.49	13	0.53	0.61	**0.46**	0.65	0.5	3.0	1.9E4	5.6	3.1	26	2.8	**1.8**
Gurobi-CORLAT	0.4	7.5	0.44	0.46	**0.38**	0.49	**0.38**	3.1	1.8E4	4.3	3.2	28	2.8	**1.8**
SCIP-CORLAT	0.39	12	0.42	0.47	**0.37**	0.51	0.39	3.1	1.8E4	7	3.2	27	3.0	**1.9**
lp_solve-CORLAT	**0.43**	14	0.47	0.49	0.49	0.56	**0.43**	3.0	1.8E4	3.8	3.1	29	**1.7**	2.3
CPLEX-RCW	0.26	0.1	0.31	0.12	0.05	0.09	**0.03**	3.1	1.2E4	4.2	3.2	26	2.8	**1.5**
CPLEX-REG	**0.39**	1.0	**0.39**	0.52	**0.39**	0.57	0.44	2.4	5.7E3	5.4	3.0	24	2.8	**1.9**
CPLEX-CR	0.46	2.3	0.58	0.47	**0.43**	0.61	0.45	**3.6**	2.7E4	8.6	6.1	32	6.5	4.3
CPLEX-CRR	0.44	0.74	0.54	0.42	0.38	0.49	**0.37**	**4.6**	4.8E4	12	9.7	36	12	6.9
LK-H-RUE	**0.62**	9.5E2	0.63	0.66	**0.62**	0.89	0.68	3.5	1.4E4	**1.0**	7.3	23	6.3	5.7
LK-H-RCE	0.72	6.5E2	0.72	0.83	**0.71**	1.0	0.77	3.7	1.6E4	**2.5**	7.5	23	6.3	5.5
LK-H-TSPLIB	6.5	16	80	1.8	1.8	1.4	**1.0**	1.7	1.4E4	2.8	0.51	4	0.31	**0.12**
Concorde-RUE	0.54	6.1E4	**0.43**	0.46	**0.43**	0.6	0.45	3.6	1.4E4	**2.5**	7.3	29	6	5.2
Concorde-RCE	**0.33**	28	0.34	0.36	0.34	0.46	0.36	3.4	1.6E4	**2**	7.4	26	6.2	5.2
Concorde-TSPLIB	3.3	32	14	1.1	1.1	0.64	**0.57**	1.7	1.4E4	3.0	0.59	4.5	0.48	**0.15**

Table 2: Quantitative comparison of models for runtime predictions on previously unseen instances. We report 2-fold cross-validation performance. Lower RMSE values are better (0 is optimal). Note the very large RMSE values for the ridge regression variants on some data sets (we use scientific notation, denoting "$\times 10^x$" as "Ex"); these large errors are due to extremely small/large predictions for a few data points.

being easy, and hard ones as being hard. Take, for example the case of predicting the runtime of Minisat 2.0 on a heterogeneous mix of SAT competition instances (see the leftmost column in Figure 4 and the top row of Table 2). Minisat 2.0 runtimes varied by almost six orders of magnitude, while predictions with the better models rarely were off by more than one order of magnitude (outliers may draw the eye in the scatter plot, but quantitatively the RMSE for predicting $\log_{10}$ runtime was low—*e.g.*, 0.55 for random forests, which means an average misprediction of about half an order of magnitude). While the models were certainly not perfect, note that even the relatively poor predictions of ridge regression variant RR enabled the portfolio-based algorithm selector SATzilla [110] to win five medals in each of the 2007 and 2009 SAT competitions. Better models, in particular random forests, improved SATzilla's

performance substantially further [111].

In our experiments, random forests were indeed the overall winner among the different methods, yielding the best predictions in terms of all our quantitative measures.[8] For SAT, they were always the best method, and for MIP they clearly yielded the best performance for the most heterogeneous instance set, BIGMIX (see Column 2 of Figure 4). We attribute the strong performance of random forests on highly heterogeneous data sets to the fact that, as a tree-based approach, they can model very different parts of the data separately; in contrast, the other methods allow the fit in a given part of the space to be influenced more by data in distant parts of the space. Indeed, all ridge regression variants (particularly RR-el) had problems with extremely-badly-predicted outliers for BIGMIX. For the more homogeneous MIP data sets, either random forests or projected processes performed best, often followed closely by ridge regression variant RR. The performance of CPLEX on set RCW was a special case in that it could be predicted extremely well by all models (see Column 3 of Figure 4). Finally, for TSP, projected processes and ridge regression had a slight edge for the homogeneous RUE and RCE benchmarks, whereas tree-based methods (once again) performed best on the most heterogeneous benchmark, TSPLIB. The last column of Figure 4 shows that, in the case where random forests performed worst, the qualitative differences in predictions were small. In terms of computational requirements, random forests were amongst the cheapest methods, taking between 0.1 and 11 seconds for learning a model.

Our original ridge regression method with backward elimination of redundant features (RR-el) performed very poorly in most cases, not eliminating enough features and suffering from overfitting. The one interesting exception was CPLEX-RCW (see Column 3 of Figure 4), where RR-el performed substantially better than the other two ridge regression methods. Due to its poor performance elsewhere and its extremely long training times (up to 3 days for the 7 012 instances in COMPETITON, compared to 46 seconds for the second-slowest method, projected processes), we did not consider RR-el in our further experiments.

6.4. Impact of Hyperparameter Optimization

Most modeling methods discussed in this paper have free hyperparameters that can be set by minimizing cross-validation loss. While, to the best of our knowledge, all previous work on runtime prediction has used fixed default parameters, we also experimented with optimizing these hyperparameters for every method in our experiments (using the gradient-free optimizer DIRECT [58] to minimize 2-fold cross-validated RMSE loss on the training set with a budget of 30 function evaluations). Table 3 shows representative results: hyperparameter optimization improved robustness somewhat for the ridge regression methods (decreasing the number of extreme outlier predictions) and improved most models slightly across the board. However, these improvements came at the expense of slowing down training by up to factors above 1 000.[9] In practice, the

[8] For brevity, we only report RMSE values in the tables here; comparative results for correlation coefficients and log likelihoods, given in Table B.1 in the online appendix, are qualitatively similar.

[9] Although we fixed the number of hyperparameter optimization steps, variation in model parameters affected learning time more for some model families than for others; for SP, slowdowns reached up to a factor of 2 000 (dataset Minisat 2.0-HAND).

	RMSE								Time to learn model (s)							
	RR		SP		NN		RF		RR		SP		NN		RF	
Domain	λ_{def}	λ_{opt}	λ_{def}	λ_{opt}	λ_{def}	λ_{opt}	λ_{def}	λ_{opt}	λ_{def}	λ_{opt}	λ_{def}	λ_{opt}	λ_{def}	λ_{opt}	λ_{def}	λ_{opt}
Minisat 2.0-COMPETITON	1.1	**0.95**	**1.3**	**1.3**	0.7	**0.68**	**0.55**	**0.55**	**4.7**	340	**15**	6707	**12**	4587	**11**	336
Minisat 2.0-SWV-IBM	**0.54**	1.9	0.75	**0.46**	**0.4**	0.45	0.21	**0.2**	**3.2**	249	**7.5**	8200	**2.6**	361	**1.5**	57
CPLEX-BIGMIX	9E11	**1.1**	1.1E7	**1.3**	1.1	**0.96**	**0.72**	**0.72**	**3**	111	**8.6**	1043	**2.7**	182	**1.9**	63
CPLEX-CORLAT	**0.49**	0.51	0.53	**0.47**	0.61	**0.54**	0.5	**0.49**	**2.8**	227	**5.7**	7806	**3.1**	281	**1.8**	49
CPLEX-RCW	0.26	**0.2**	0.31	**0.16**	**0.12**	**0.12**	**0.03**	**0.03**	**2.8**	259	**4.4**	7608	**3**	197	**1.5**	52
LK-H-TSPLIB	**6.5**	11	80	**1.1**	1.8	**1.7**	**1**	1.1	**1.4**	54	**3**	224	**0.53**	44	**0.13**	4.7
Concorde-RUE	0.54	**0.41**	0.43	**0.42**	0.46	**0.42**	**0.45**	**0.45**	**3.3**	173	**2.7**	3447	**7.2**	322	**5**	134

Table 3: Quantitative evaluation of the impact of hyperparameter optimization on predictive accuracy. For each model family with hyperparameters, we report performance achieved with and without hyperparameter optimization. We compare 10-fold cross-validation performance for the default and for hyperparameters optimized using DIRECT with 2-fold cross-validation, using bold-face to indicate better results. Tables B.2 and B.3 (in the online appendix) provide results for all benchmarks.

small improvements in predictive performance that can be obtained via hyperparameter optimization are likely not to justify this drastic increase in computational cost (e.g., consider model-based algorithm configuration procedures, which iterate between model construction and data gathering, constructing thousands of models during typical algorithm configuration runs [50]). Thus, we evaluate model performance based on fixed default hyperparameters in the rest of this article. For completeness, the online appendix reports analogous results for models with optimized hyperparameters.

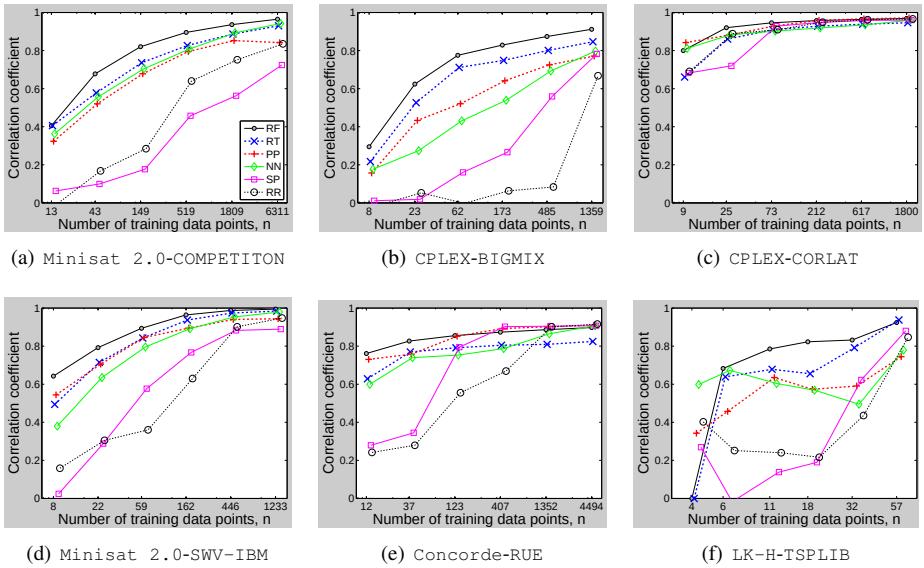
6.5. Predictive Quality with Sparse Training Data

We now study how the performance of EPM techniques changes based on their amounts of training data. Figure 5 visualizes this functional dependency for six representative benchmarks; data for all benchmarks appears in our online appendix. Here and in the following, we use CC rather than RMSE for such scaling plots, for two reasons. First, RMSE plots are often cluttered due to outlier instances for which prediction accuracy is poor (these mostly occur for the ridge regression methods). Second, plotting CC allows immediate visual performance comparisons across benchmarks, since CC $\in [-1, 1]$.

Overall, random forests performed best across training set sizes. Both versions of ridge regression (SP and RR) performed poorly for small training sets. This observation is significant, since most past work employed ridge regression to construct empirical performance models in situations when data was sparse as, *e.g.*, in `SATzilla` [110].

7. Performance Predictions for New Parameter Configurations

We now move from considering the ability of different models to predict an algorithm's runtime on new *instances* to the prediction of algorithm performance given new *parameter settings* (or *configurations*). To study this issue in isolation, we begin in this section by considering only single problem instances, but varying algorithm parameters. (Note that this means that we have no use for instance features here.) We consider parameter configurations and instance features jointly in Section 8.



(a) Minisat 2.0-COMPETITON (b) CPLEX-BIGMIX (c) CPLEX-CORLAT

(d) Minisat 2.0-SWV-IBM (e) Concorde-RUE (f) LK-H-TSPLIB

Figure 5: Prediction quality for varying numbers of training instances. For each model and number of training instances, we plot the mean (taken across 10 cross-validation folds) correlation coefficient (CC) between true and predicted runtimes for new test instances; larger CC is better, 1 is perfect. Figures B.11–B.13 (in the online appendix) show equivalent plots for the other benchmarks.

Algorithm	Parameter type	# parameters of this type	# values considered	Total # configurations
SPEAR	Categorical	10	2–20	8.34×10^{17}
	Integer	4	5–8	
	Continuous	12	3–6	
CPLEX	Boolean	6	2	1.90×10^{47}
	Categorical	45	3–7	
	Integer	18	5–7	
	Continuous	7	5–8	

Table 4: Algorithms and characteristics of their parameter configuration spaces.

7.1. Parameter Configuration Spaces

Here and in Section 8, we study two highly parametric algorithms for two different domains: SPEAR for SAT and CPLEX for MIP. We now provide some details on these algorithms and their parameters.

SPEAR is a state-of-the-art SAT solver for industrial instances that was developed by Babić [3]. With appropriate parameter settings, it was shown to be the best available solver for certain types of SAT-encoded hardware and software verification instances [45] (the same IBM and SWV instances we use here). It also won the quantifier-free bit-vector arithmetic category of the 2007 Satisfiability Modulo Theories Competition. For our experimental study, we used exactly the same parameter configuration space as in previous work [45]. This includes 26 parameters, out of which ten are categorical, four are integral, and twelve are continuous. The categorical parameters mainly control

heuristics for variable and value selection, clause sorting, resolution ordering, and also enable or disable optimizations, such as the pure literal rule. The continuous and integer parameters mainly deal with activity, decay, and elimination of variables and clauses, as well as with the randomized restart interval and percentage of random choices; we discretized each of them to between three and eight values. In total, and based on our discretization of continuous parameters, SPEAR has 8.34×10^{17} different configurations.

IBM ILOG CPLEX is the most-widely used commercial optimization tool for solving MIPs; it is used by over 1 300 corporations (including a third of the Global 500) and researchers at more than 1 000 universities [55]. CPLEX exposes many parameters whose settings impact its performance. We used the same configuration space with 76 parameters as in previous work [47]. These parameters exclude all CPLEX settings that change the problem formulation (*e.g.*, the optimality gap below which a solution is considered optimal). They include 12 preprocessing parameters (mostly categorical); 17 MIP strategy parameters (mostly categorical); 11 categorical parameters deciding how aggressively to use which types of cuts; 9 real-valued MIP "limit" parameters; 10 simplex parameters (half of them categorical); 6 barrier optimization parameters (mostly categorical); and 11 further parameters. In total, and based on our discretization of continuous parameters, these parameters gave rise to 1.90×10^{47} unique configurations.

7.2. Experimental Setup

For the experiments in this and the next section, we gathered a large amount of runtime data for these solvers by executing them with different configurations on different instances. Specifically, we evaluated models of each solver on distributions for which we expected the solver to yield state-of-the-art performance: SPEAR for distributions SWV and IBM; CPLEX for all MIP distributions discussed in the previous section. For each combination of solver and instance distribution, we measured the runtime of each of $M = 1\,000$ randomly-sampled parameter configurations on each of the P problem instances available for the distribution, with P ranging from 604 to 2 000. The resulting runtime observations can be thought of as a $M \times P$ matrix. Since gathering this runtime matrix meant performing $M \cdot P$ (*i.e.*, between 604 000 and 2 000 000) runs per data set, we used a large cluster (the 840-node Westgrid cluster Glacier, each of whose nodes is equipped with two 3.06 GHz Intel Xeon 32-bit processors and 2–4GB RAM) and gave each single algorithm run "only" a cutoff time of 300 seconds (compared to the 3 000 seconds for the runs with the default parameter setting in Section 7). The data collection took over 60 CPU years, with time requirements for individual data sets between 1.3 CPU years (SPEAR on SWV, where many runs took less than a second) and 18 CPU years (CPLEX on RCW, where most runs timed out).

We pause to note that, while a sound empirical evaluation of our methods for combined predictions in the instance and configuration space required gathering this very large amount of data, such extensive experimentation is not required to use them in practice. Indeed, as we will demonstrate in Section 8.3, our methods often yield surprisingly accurate predictions based on data that can be gathered overnight on a single machine.

In this section, we consider the performance of EPMs as parameters vary but instance features do not; thus, we used only one instance from each distribution. For each dataset, we selected the easiest benchmark instance amongst the ones for which the default

	RMSE						Time to learn model (s)					
Domain	RR	SP	NN	PP	RT	RF	RR	SP	NN	PP	RT	RF
CPLEX-BIGMIX	0.26	0.34	0.4	**0.24**	0.33	0.25	5.4	**0.84**	3.9	36	4.3	3
CPLEX-CORLAT	0.56	0.67	0.76	**0.53**	0.75	0.55	5.6	**2.6**	3.8	36	4.2	3
CPLEX-REG	0.43	0.5	0.6	0.42	0.49	**0.38**	5.4	**2.2**	3.7	30	3.9	2.9
CPLEX-RCW	**0.2**	0.25	0.3	0.21	0.28	0.21	5.5	**0.53**	3.7	38	2.2	2
SPEAR-IBM	**0.25**	0.75	0.74	**0.25**	0.31	0.28	3.1	**0.25**	2.7	13	1.6	1.5
SPEAR-SWV	0.36	0.52	0.59	**0.35**	0.41	0.36	2.8	**0.22**	2.6	13	1.6	1.5

Table 5: Quantitative comparison of models for runtime predictions on previously unseen parameter configurations. We report 10-fold cross-validation performance. Lower RMSE is better (0 is optimal). Table B.5 (in the online appendix) provides additional results (correlation coefficients and log likelihoods).

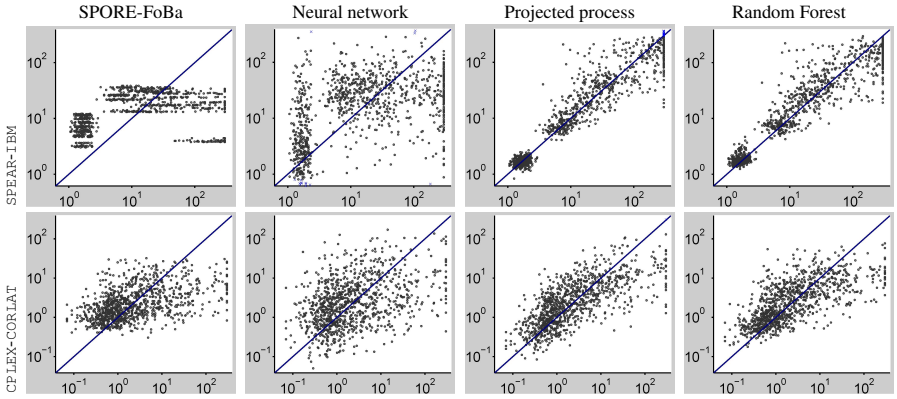
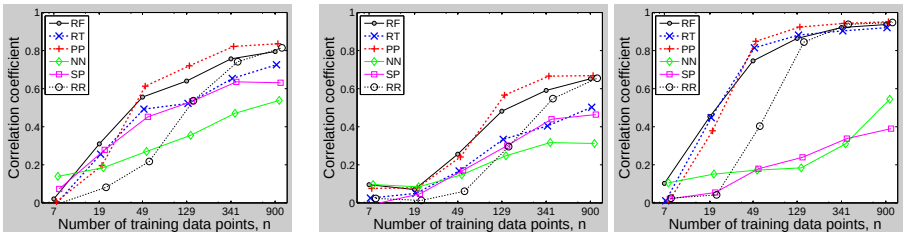


Figure 6: Visual comparison of models for runtime predictions on previously unseen parameter configurations. In each scatter plot, the x-axis denotes true runtime and the y-axis cross-validated runtime as predicted by the respective model. Each dot represents one parameter configuration. Figures B.14 and B.15 (in the online appendix) provide results for all domains and also show the performance of regression trees and ridge regression variant RR (whose predictions were similar to random forests and projected processes, with somewhat larger spread for regression trees).

parameter configuration required more than ten seconds on our reference machines. As before, we used 10-fold cross validation to judge the accuracy of our model predictions for previously unseen parameter configurations.

7.3. Predictive Quality

Table 5 quantifies the performance of all models on all benchmark domains, and Figure 6 visualizes predictions. Again, we see that qualitatively, solver runtime as a function of parameter settings could be predicted quite well by most methods, even as runtimes varied by factors of over 1 000 (see Figure 6). We observe that projected processes, random forests, and ridge regression variant RR consistently outperformed regression trees; this is significant, as regression trees are the only model that has previously been used for predictions in configuration spaces with categorical parameters [8]. On the other hand, the poor performance of neural networks and of ridge regression variant SPORE-FoBa (which mainly differs from variant RR in its feature expansion and selection) underlines that selecting the right (combinations of) features is



(a) CPLEX-BIGMIX (b) CPLEX-CORLAT (c) SPEAR-IBM

Figure 7: Quality of predictions in the configuration space, as dependent on the number of training configurations. For each model and number of training instances, we plot mean ± standard deviation of the correlation coefficient (CC) between true and predicted runtimes for new test configurations. Figure B.16 (in the online appendix) shows equivalent results for all benchmarks.

not straightforward. Overall, the best performance was achieved by projected processes (applying our new kernel function for categorical parameters from Section 4.1.2). As in the previous section, however, random forests were also either best or very close to the best for every data set.

7.4. Predictive Quality with Sparse Training Data

Results remained similar when varying the number of training configurations. As Figure 7 shows, projected processes performed best overall, closely followed by random forests. Ridge regression variant RR often produced poor predictions when trained using a relatively small number of training data points, but performed well when given sufficient data. Finally, both ridge regression variant SPORE-FoBa and neural networks performed relatively poorly regardless of the amount of data given.

8. Performance Predictions in the Joint Space of Instance Features and Parameter Configurations

We now move to the most general—and challenging—problem: predicting an algorithm's runtime with a previously unseen parameter configuration and on a previously unseen problem instance. We evaluated EPMs in this setting using the same algorithms and instance distributions as in Section 7.

8.1. Experimental Setup

We used the same runtime matrix data for SPEAR and CPLEX described in Section 7.2: for each benchmark, this includes the runtimes for M configurations on each of P instances. We split both the M configurations and the P instances into training and test sets of equal size (using uniform random permutations). We then trained our EPMs on a fixed number of n randomly-selected combinations of the $P/2$ training instances and $M/2$ training configurations. In the first experiments discussed here (Sections 8.2 and 8.3) we tested predictions on both previously unseen configurations and instances. Later in the section (Section 8.4) we evaluate predictions made on all four combinations of training/test instances and training/test configurations.

	RMSE						Time to learn model (s)					
Domain	RR	SP	NN	PP	RT	RF	RR	SP	NN	PP	RT	RF
CPLEX-BIGMIX	$> 10^{100}$	4.5	0.68	0.78	0.74	**0.55**	**25**	34	49	84	52	47
CPLEX-CORLAT	0.53	0.57	0.56	0.53	0.67	**0.49**	**26**	27	52	76	46	40
CPLEX-REG	**0.17**	0.19	0.19	0.19	0.24	**0.17**	23	**14**	50	77	32	31
CPLEX-RCW	0.1	0.12	0.12	0.12	0.12	**0.09**	24	**13**	45	78	25	24
CPLEX-CR	0.41	0.43	0.42	0.42	0.52	**0.38**	**26**	37	54	88	47	43
CPLEX-CRR	0.35	0.37	0.37	0.39	0.43	**0.32**	**29**	35	48	81	38	37
SPEAR-IBM	0.58	11	0.54	0.52	0.57	**0.44**	**15**	31	41	70	36	30
SPEAR-SWV	0.58	0.61	0.63	0.54	0.55	**0.44**	**15**	42	41	69	42	28
SPEAR-SWV-IBM	0.65	0.69	0.65	0.65	0.59	**0.45**	**17**	35	39	70	41	32

Table 6: Quantitative comparison of models for runtime predictions on unseen instances and configurations. For CPLEX-BIGMIX, RR had a few extremely poorly predicted outliers, with the maximal prediction of $\log_{10}$ runtime exceeding 10^{100} (*i.e.*, a runtime prediction above $10^{10^{100}}$); thus, we can only bound its RMSE from below. Models were based on 10 000 data points. Table B.6 (in the online appendix) provides additional results (correlation coefficients and log likelihoods).

8.2. Predictive Quality

We first focus on the most interesting case, predictions for previously unseen test instances and configurations. Table 6 provides quantitative results of model performance based on $n = 10\,000$ training data points, and Figure 8 visualizes performance. On the highest level, we note that the best models generalized to new configurations *and* to new instances almost as well as to either alone (compare Sections 6 and 7, respectively). On the most heterogeneous data set CPLEX-BIGMIX, we once again witnessed extremely poorly predicted outliers for the ridge regression variants, but in all other cases, the models captured the large spread in runtimes (above 5 orders of magnitude) quite well. As in the experiments in Section 6.3, the tree-based approaches, which are able to model different regions of the input space independently, performed best on the most heterogeneous data sets. Figure 8 also shows some qualitative differences in predictions: for example, ridge regression, neural networks, and projected processes sometimes overpredicted the runtime of the shortest runs, while the tree-based methods did not have this problem. Random forests performed best in all cases, consistently with their robust predictions in both the instance and the configuration space observed earlier.

8.3. Predictive Quality with Sparse Training Data

Next, we studied the amount of data that was actually needed to obtain good predictions, varying the number n of randomly selected combinations of training instances and configurations. Figure 9 shows the correlation coefficients achieved by the various methods as a function of the amount of training data. Overall, we note that most models already performed remarkably well (yielding correlation coefficients of 0.9 and higher) based on a few hundred training data points. This confirmed the practicality of our methods: on a single machine, it takes at most 12.5 hours to execute 150 algorithm runs with a cutoff time of 300 seconds. Thus, even users without access to a cluster can expect to be able to execute sufficiently many algorithm runs overnight to build a decent empirical performance model for their algorithm and instance distribution of interest. Examining our results in some more detail, the ridge regression variants again had trouble on the most heterogeneous benchmark CPLEX-BIGMIX, but otherwise

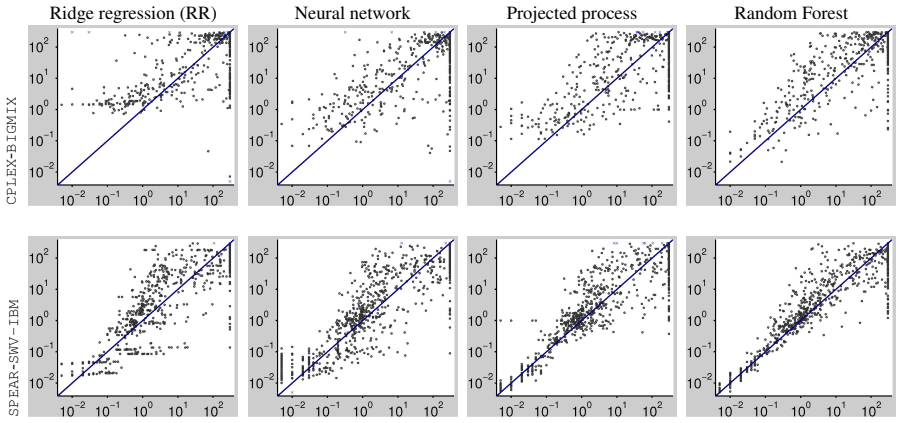
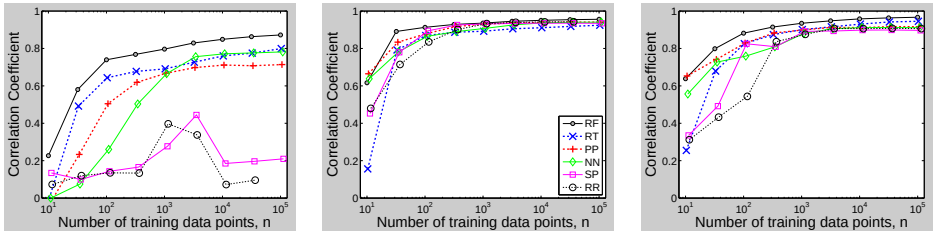


Figure 8: Visual comparison of models for runtime predictions on pairs of previously unseen test configurations and instances. In each scatter plot, the x-axis shows true runtime and the y-axis cross-validated runtime as predicted by the respective model. Each dot represents one combination of an unseen instance and parameter configuration. Figures B.17–B.19 (in the online appendix) include all domains and also show the performance of SPORE-FoBa (very similar to RR) and regression trees (similar to RF, somewhat larger spread).



(a) CPLEX-BIGMIX (b) CPLEX-CORLAT (c) SPEAR-SWV-IBM

Figure 9: Quality of predictions in the joint instance/configuration space as a function of the number of training data points. For each model and number of training data points, we plot mean correlation coefficients between true and predicted runtimes for new test instances and configurations. We omit standard deviations to avoid clutter, but they are very high for the two ridge regression variants. Figure B.20 (in the online appendix) shows corresponding, and qualitatively similar, results for all benchmarks.

performed quite well. Overall, random forests performed best across different training set sizes. Naturally, all methods required more data to make good predictions for heterogeneous benchmarks (*e.g.*, CPLEX-BIGMIX) than for relatively homogeneous ones (*e.g.*, CPLEX-CORLAT, for which a remarkably low number of 30 data points already yielded correlation coefficients exceeding 0.9).

8.4. Evaluating Generalization Performance in Instance and Configuration Space

So far we have evaluated model accuracy for both new instances and new parameter configurations. However, other scenarios are also important in practice:

1. **Predictions for training configurations on training instances.** Predictions for this most basic case are useful for succinctly modeling known algorithm

Domain	Instances	Training configurations						Test configurations					
		RR	SP	NN	PP	RT	RF	RR	SP	NN	PP	RT	RF
CPLEX-BIGMIX	Training	0.6	0.6	0.55	0.65	0.59	**0.43**	0.6	0.6	0.56	0.65	0.62	**0.45**
	Test	$> 10^{100}$	4.5	0.67	0.78	0.71	**0.54**	$> 10^{100}$	4.5	0.68	0.78	0.74	**0.55**
CPLEX-CORLAT	Training	0.5	0.55	0.47	0.49	0.54	**0.39**	0.52	0.56	0.54	0.51	0.64	**0.46**
	Test	0.51	0.55	0.5	0.51	0.58	**0.42**	0.53	0.57	0.56	0.53	0.67	**0.49**
CPLEX-REG	Training	0.15	0.18	0.15	0.16	0.17	**0.12**	**0.16**	0.18	0.18	0.17	0.22	**0.16**
	Test	0.17	0.19	0.17	0.18	0.19	**0.14**	**0.17**	0.19	0.19	0.19	0.24	**0.17**
CPLEX-RCW	Training	0.09	0.11	0.09	0.1	0.08	**0.06**	0.1	0.12	0.11	0.11	0.12	**0.09**
	Test	0.09	0.12	0.09	0.11	0.08	**0.06**	0.1	0.12	0.12	0.12	0.12	**0.09**
CPLEX-CR	Training	0.39	0.41	0.37	0.4	0.45	**0.32**	0.4	0.42	0.41	0.41	0.49	**0.36**
	Test	0.4	0.42	0.38	0.41	0.47	**0.34**	0.41	0.43	0.42	0.42	0.52	**0.38**
CPLEX-CRR	Training	0.33	0.35	0.33	0.36	0.38	**0.28**	0.34	0.36	0.36	0.37	0.41	**0.31**
	Test	0.34	0.37	0.34	0.38	0.4	**0.29**	0.35	0.37	0.37	0.39	0.43	**0.32**
SPEAR-IBM	Training	0.57	0.64	0.5	0.48	0.43	**0.34**	0.57	0.64	0.51	0.48	0.45	**0.36**
	Test	0.57	11	0.53	0.52	0.57	**0.42**	0.58	11	0.54	0.52	0.57	**0.44**
SPEAR-SWV	Training	0.52	0.56	0.56	0.46	0.37	**0.3**	0.52	0.56	0.57	0.47	0.43	**0.34**
	Test	0.57	0.61	0.62	0.53	0.51	**0.4**	0.58	0.61	0.63	0.54	0.55	**0.44**
SPEAR-SWV-IBM	Training	0.63	0.66	0.61	0.62	0.48	**0.36**	0.63	0.66	0.61	0.62	0.5	**0.38**
	Test	0.64	0.69	0.64	0.65	0.58	**0.43**	0.65	0.69	0.65	0.65	0.59	**0.45**

Table 7: Root mean squared error (RMSE) obtained by various empirical performance models for predicting the runtime based on combinations of paramater configurations and intance features. We trained on 10 000 randomly-sampled combinations of training configurations and instances, and report performance for the four combinations of training/test instances and training/test configurations.

performance data. Interestingly, several methods already perform poorly here.

2. **Predictions for test configurations on training instances.** This case is important in algorithm configuration, where the goal is to find high-quality parameter configurations for the given training instances [50, 51].
3. **Predictions for training configurations on test instances.** Such predictions can be used to make a per-instance decision about which of a set of given parameter configurations will perform best on a previously unseen test instance [107, 61, 94].
4. **Predictions for test configurations on test instances.** This most general case, addressed also by [85, 19] and already investigated above, is the most natural "pure prediction" problem. It is also important for per-instance algorithm configuration, where one could use a model to search for the configuration that is most promising for a previously-unseen test instance [46].

Our results are summarized in Table 7 and Figures 10 and 11. For the figures, we sorted instances by average hardness (across configurations), and parameter configurations by average performance (across instances), generating a heatmap with instances on the x-axis, configurations on the y-axis, and greyscale values representing algorithm runtime for given configuration/instance combinations. We compare heatmaps representing true runtimes against those based on the predictions obtained from each of our models. Here, we only show results for the two domains where the performance advantage of random forests (the overall best method based on our results reported so far) over the other methods was highest (the heterogeneous data set SPEAR-SWV-IBM) and lowest (the homogeneous data set CPLEX-CORLAT); heatmaps for all data sets and model types are

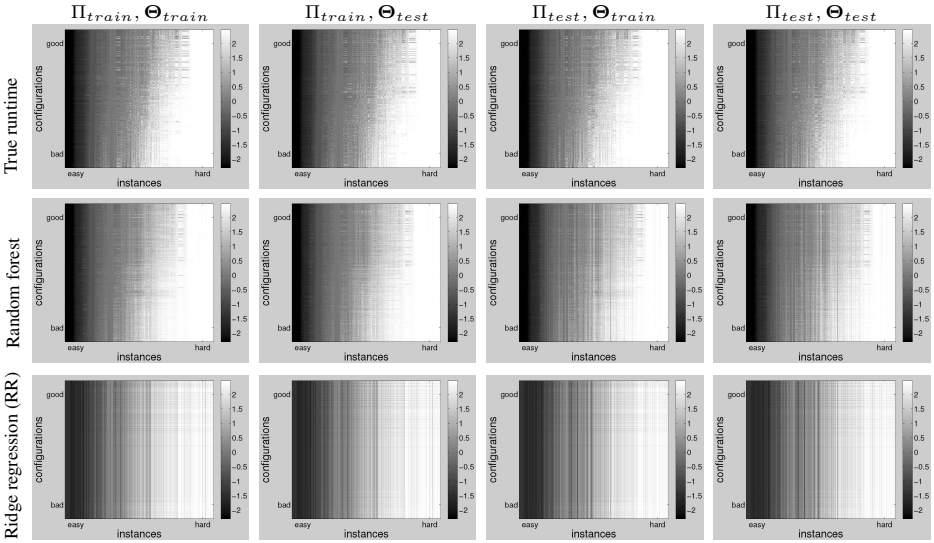


Figure 10: True and predicted runtime matrices for dataset SPEAR-SWV-IBM, for all combinations of training/test instances (Π_{train} and Π_{test}, respectively) and training test configurations ($\mathbf{\Theta}_{train}$ and $\mathbf{\Theta}_{test}$, respectively). (Plots for all models and benchmarks are given in Figures B.21–B.29, in the online appendix.) The predicted matrix of regression trees is visually indistinguishable from that of random forests, and those of all other methods closely resemble that of ridge regression.

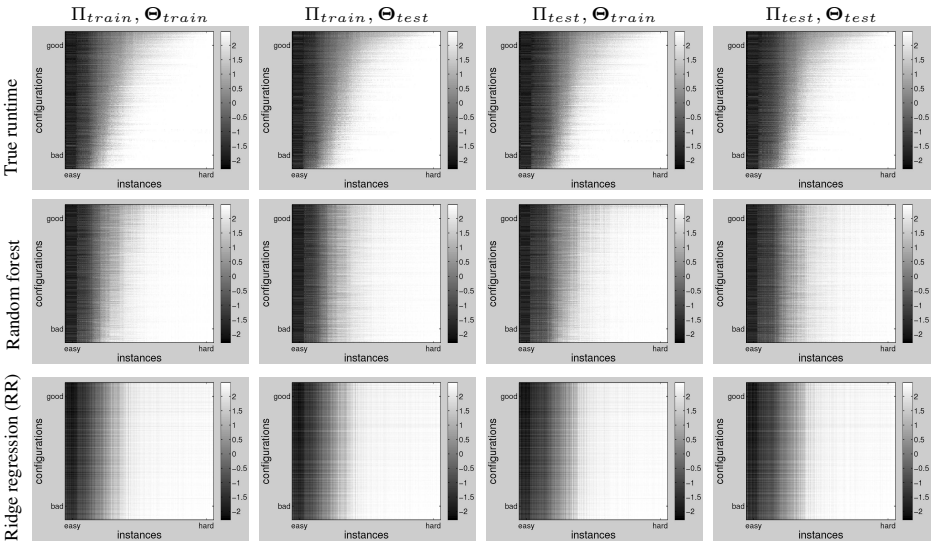


Figure 11: Same type of data as in Figure 10 but for dataset CPLEX-CORLAT.

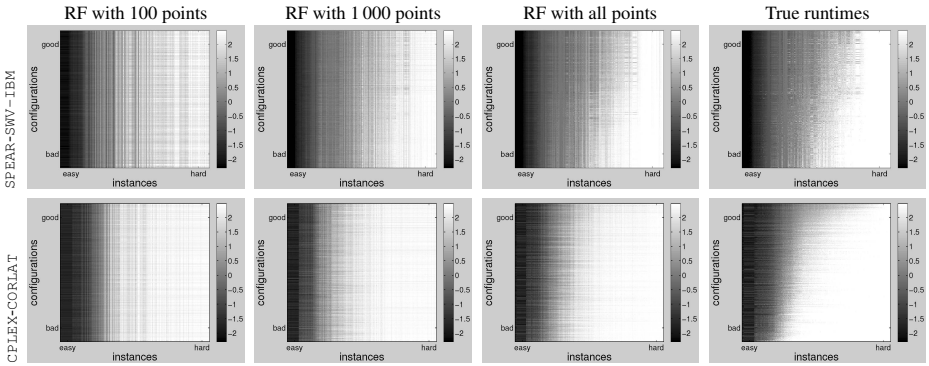


Figure 12: Predicted runtime matrices with different number of training data points, compared to true runtime matrix. "All points" means the entire crossproduct of training instances and training configurations (342 500 data points for SPEAR-SWV-IBM and 500 000 for CPLEX-CORLAT). (Plots for all benchmarks are given in Figure B.30 in the online appendix.)

given in Figures B.11–B.19 in the online appendix.

Figure 10 shows the results for benchmark SPEAR-SWV-IBM. It features one column for each of the four combinations of training/test instances and training/test configurations, allowing us to assess visually how well the respective generalization works for each of the models. We note that in this case, the true heatmaps are almost indistinguishable from those predicted by random forests (and regression trees). Even for the most challenging case of unseen problem instances and parameter configurations, the tree-based methods captured the non-trivial interaction pattern between instances and parameter configurations. On the other hand, the non-tree based methods (ridge regression variants, neural networks, and projected processes) only captured instance hardness, failing to distinguish good from bad configurations even in the simplest case of predictions for training instances and training configurations.

Figure 11 shows the results for benchmark CPLEX-CORLAT. For the simplest case of predictions on training instances and configurations, the tree-based methods yielded predictions close to the true runtimes, capturing both instance hardness and performance of parameter configurations. In contrast, even in this simple case, the other methods only captured instance hardness, predicting all configurations to be roughly equal in performance. Random forests generalized better to test instances than to test configurations (compare the third and second columns of Figure 11); this trend is also evident quantitatively in Table 7 for all CPLEX benchmarks. We note that regression tree predictions were visually indistinguishable from those of random forests; this strong qualitative performance is remarkable, considering that quantitatively they performed worse than other methods in terms of measures such as RMSE (see the results for CPLEX-CORLAT in Table 7).

Finally, we investigated once more how predictive quality depends on the amount of training data, focusing on our best-performing method, random forests (Figure 12). For SPEAR-SWV-IBM, 100 training data points already sufficed to obtain random forest models that captured the most salient features (*e.g.*, they correctly determined the simplicity

of the roughly 20% easiest instances); more training data points gradually improved qualitative predictions, especially in distinguishing good from bad configurations. Likewise, for CPLEX-CORLAT, salient features (*e.g.*, the simplicitity of the roughly 25% easiest instances) could already be detected based on 100 training data points, and more training data improved qualitative predictions to capture some of the differences between good and bad configurations. Overall, increases in the training set size yielded diminishing returns, and even predictions based on the entire cross-product of training instances and parameter configurations were not much different from those based on a subset of 10 000 samples.

9. An Improved Mechanism for Handling Censored Runtimes in Random Forests

Most past work on predicting algorithm runtime treated algorithm runs that were terminated prematurely at a so-called *captime* κ as if they finished at time κ. Thus, we adopted the same practice in the model comparisons we have described so far (using captimes of 3 000 seconds for the runs in Section 6 and 300 seconds for the runs in Sections 7 and 8). Now, we revisit this issue in the context of our best-performing model, random forests.

Formally, terminating an algorithm run after a captime (or *censoring threshold*) κ yields a *right-censored* data point: we only learn that κ is a lower bound on the actual time the algorithm run required. Let y_i denote the *actual* (unknown) runtime of algorithm run i. Under partial right-censoring, our training data is $(\mathbf{x}_i, z_i, c_i)_{i=1}^n$, where $\mathbf{x}_i$ is our usual input vector (a vector of instance features, parameter values, or both combined), $z_i \in \mathbb{R}$ is a (possibly censored) runtime observation, and $c_i \in \{0, 1\}$ is a censoring indicator such that $z_i = y_i$ if $c_i = 0$ and $z_i < y_i$ if $c_i = 1$.

Observe that the typical, simplistic strategy for dealing with censored data produces biased models; intuitively, treating slow runs as though they were faster than they really were biases our training data downwards, and hence likewise biases predictions. Statisticians, mostly in the literature on so-called "survival analysis" from actuarial science, have developed strategies for building unbiased regression models based on censored data [77]. (Actuaries need to predict when people will die, given mortality data and the ages of people still living.) Gagliolo et al. [26, 25] were the first to use techniques from this literature for runtime prediction. Specifically, they used a method for handling censored data in parametric probabilistic models and employed the resulting models to construct dynamic algorithm portfolios. In the same survival analysis literature, Schmee & Hahn [89] described an iterative procedure for handling censored data points in linear regression models. We employed this technique to improve the runtime predictions made by our portfolio-based algorithm selection method SATzilla [108]. While to the best of our knowledge, no other methods from this literature have been applied to algorithm runtime prediction, there exist several candidates for future consideration. In Gaussian processes, one could use approximations to handle the non-Gaussian observation likelihoods resulting from censorship; for example, Ertin [24] described a Laplace approximation for handling right-censored data. Random forests (RFs) have previously been adapted to handle censored data [91, 41], but the classical methods yield non-parametric Kaplan–Meier estimators that are undefined beyond the largest

uncensored data point. Here, we introduce a simple improvement of the method by Schmee & Hahn [89] for use with random forests.

We denote the PDF and CDF of a Normal distribution by φ and Φ, respectively. Let $\mathbf{x}_i$ be an input for which we observed a censored runtime of κ_i. Given a Gaussian predictive distribution $\mathcal{N}(\mu_i, \sigma_i^2)$, the truncated Gaussian distribution $\mathcal{N}(\mu_i, \sigma_i^2)_{\geq \kappa_i}$ is defined by the PDF

$$p(y) = \begin{cases} 0 & y < \kappa_i \\ \frac{1}{\sigma_i} \cdot \varphi(\frac{x-\mu_i}{\sigma_i}) / (1 - \Phi(\frac{\mu_i - \kappa_i}{\sigma_i})) & y \geq \kappa_i. \end{cases}$$

Our algorithm is inspired by the approach of Schmee and Hahn [89], which is an Expectation Maximization (EM) algorithm. Applied to an RF model as its base model, that algorithm would first fit an initial RF using only uncensored data and then iterate between the following E and M steps:

(E) For each tree T in the RF and each i s.t. $c_i = 1$: $\hat{y}_i^{(T)} \leftarrow$ mean of $\mathcal{N}(\mu_i, \sigma_i^2)_{\geq z_i}$;

(M) Refit the RF using $\left(\boldsymbol{\theta}_i, \hat{y_i}^{(T)}\right)_{i=1}^{n}$ as the basis for tree T.

Here, $\mathcal{N}(\mu_i, \sigma_i^2)_{\geq z_i}$ in the E step denotes the predictive distribution of the *current* RF for data point i. While the mean of $\mathcal{N}(\mu_i, \sigma_i^2)_{\geq z_i}$ is the best single value to impute for the ith data point, in the context of RF models this approach yields overly confident predictions: all trees agree on the predictions for censored data points. To preserve our uncertainty about the true runtime of censored runs, we change the E step to:

(E') For each tree T in the RF and each i s.t. $c_i = 1$: $\hat{y}_i^{(T)} \leftarrow$ sample from $\mathcal{N}(\mu_i, \sigma_i^2)_{\geq z_i}$.

This modified version takes our prior uncertainty into account when computing the posterior predictive distribution, thereby avoiding overly confident predictions. As an implementation detail, we note that here we do not aim to make predictions beyond a maximal time of $\kappa_{max} = 300$ seconds, and that we thus ensure that the mean imputed value does not exceed κ_{max}.[10]

9.1. Experimental Setup

We now experimentally compare Schmee & Hahn's procedure and our modified version to two baselines: ignoring censored data points altogether and treating data points that were censored at the captime κ as uncensored data points with runtime κ. We only report results for the most interesting case of predictions for previously unseen parameter configurations and instances. We used the 9 benchmark distributions from Section 8, artificially censoring the training data at different thresholds below the actual threshold. We experimented with two different types of capped data: (1) data with a fixed censoring threshold across all data points, and (2) data in which the thresholds were instance-specific (specifically, we set the threshold for all runs on an instance to

[10] In Schmee & Hahn's algorithm, this simply means imputing $\min\{\kappa_{max}, \text{mean}(\mathcal{N}(\mu_i, \sigma_i^2)_{\geq z_i})\}$. In our sampling version, it amounts to keeping track of the mean m_i of the imputed samples for each censored data point i and subtracting $m_i - \kappa_{max}$ from each sample for data point i if $m_i > \kappa_{max}$.

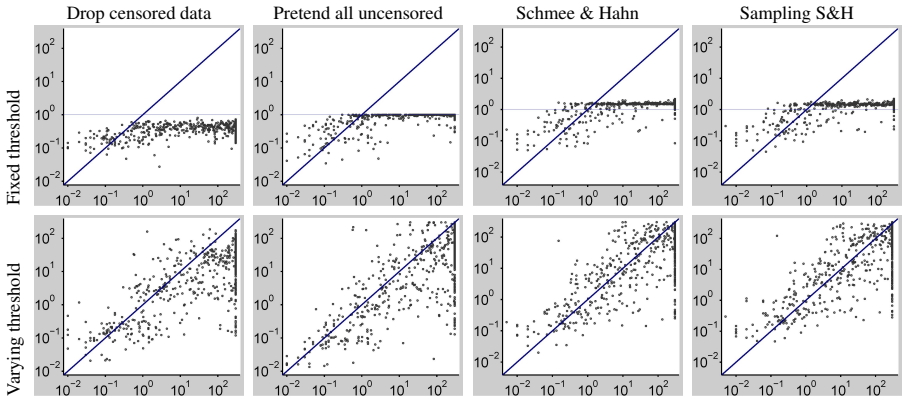


Figure 13: True and predicted runtime of various ways of handling censored data in random forests, for scenario CPLEX-BIGMIX with fixed censoring threshold of one second during training (top) and varying threshold (bottom). In each scatter plot, the x-axis indicates true runtime and the y-axis cross-validated runtime as predicted by the respective model. Each dot represents one instance. Analogous figures for all benchmarks are given in Figures B.31–B.39 (in the online appendix).

the runtime of the best of the 1 000 configurations on that instance, multiplied by a slack factor). The fixed threshold represents the sort of data generated by experimental studies like those from the previous sections of this paper, while the instance-specific threshold models practical applications of EPMs in model-based algorithm configuration procedures [49]. For both types of capped data and for all prediction strategies, we measured both predictive error (using RMSE as in the rest of the paper) and the quality of uncertainty estimates (using log-likelihood, LL, as defined in Section 6.2) on the uncensored part of the test data.

9.2. Experimental Results

Figure 13 illustrates the raw predictions for one benchmark, demonstrating the qualitative differences between the four methods for treating capped data. In the case of a fixed censoring threshold κ, simply dropping censored data yielded consistent underestimates (see the top-left plot of Figure 13), while treating censored data as uncensored at κ yielded good predictions up to κ (but not above); this strategy is thus reasonable when no predictions beyond κ are required (which is often the case; *e.g.*, throughout the main part of this article). The Schmee & Hahn variants performed similarly up to κ, but yielded unbiased predictions up to about two times κ.

Table 8 quantifies performance for all 9 benchmarks. The top left part of this table shows that dropping censored data yielded the worst prediction errors, treating this data as uncensored improved results, and using the Schmee & Hahn variants yielded further improvements. The top right part shows that for fixed thresholds, dropping censored values often yielded better uncertainty estimates than the other variants. This is because the Schmee & Hahn variants imputed similar values for all censored data points (close to the fixed threshold), yielding too little variation across trees, and thus overconfident predictions. In contrast, as shown in the bottom half of Table 8, for data with a varying captime, the Schmee & Hahn variants (in particular our new variant)

	Domain	RMSE				Log likelihood			
		Drop cens	Pretend uncens	S&H	Sampling S&H	Drop cens	Pretend uncens	S&H	Sampling S&H
fixed threshold of 1s	CPLEX-BIGMIX	2.6	2.2	**2**	**2**	**-57**	-230	-195	-192
	CPLEX-CORLAT	2.1	1.9	**1.8**	**1.8**	**-68**	-179	-151	-151
	CPLEX-CR	2.9	2.2	**2**	**2**	**-38**	-235	-192	-187
	CPLEX-CRR	3.1	2.3	**2.1**	2.1	**-42**	-254	-213	-211
	SPEAR-IBM	1.9	1.8	**1.6**	1.7	-141	-154	-132	**-131**
	SPEAR-SWV	1.4	1.2	**1.2**	**1.2**	**-37**	-75	-62	-61
	SPEAR-SWV-IBM	1.6	1.5	**1.4**	**1.4**	-108	-109	**-90**	-91
varying threshold	CPLEX-BIGMIX	0.95	0.90	0.67	**0.65**	**-1.9**	-8.7	-3.4	**-1.9**
	CPLEX-CORLAT	1.0	1.0	**0.72**	**0.72**	-4.3	-15	-6.3	**-4.2**
	CPLEX-REG	**0.20**	0.79	**0.20**	**0.20**	**-0.65**	-11	**-0.65**	-0.68
	CPLEX-RCW	0.34	0.65	**0.25**	**0.25**	**0.61**	-19	-1.4	0.38
	CPLEX-CR	0.64	0.76	**0.50**	**0.50**	-1.7	-6.8	-1.9	**-1.2**
	CPLEX-CRR	0.48	0.65	**0.39**	**0.39**	-0.73	-5.5	-1.1	**-0.56**
	SPEAR-IBM	0.73	0.67	0.58	**0.57**	-4.80	-11	-4.6	**-3.1**
	SPEAR-SWV	1.1	0.93	**0.82**	0.83	**-15**	-32	-19	-18
	SPEAR-SWV-IBM	0.83	0.78	0.65	**0.63**	-6.40	-15	-4.6	**-2.5**

Table 8: Quantitative comparison of model performance with fixed censoring threshold of one second for training data (top half) and varying threshold (using the best achieved time for each instance, *i.e.*, slack factor 1; bottom half). "S&H" is the method by Schmee & Hahn [89], "Sampling S&H" is our modified version, "Drop cens" means dropping censored data, and "Pretend uncens" means treating all observations as uncensored. (For benchmarks CPLEX-REG and CPLEX-RCW, with the common fixed threshold of one second, *all* training data was censored and predictions were thus meaningless.)

yielded competitive uncertainty estimates and clearly achieved the lowest prediction error. Furthermore, pretending censored data to be uncensored performed poorly for data with varying captimes.

Figure 14 illustrates how the quality of the different methods varied with how aggressively the data was capped. In brief, all results just presented were robust with respect to this level of aggressiveness; in particular, the Schmee & Hahn variants always yielded the lowest prediction error and, for instance-specific capping thresholds, typically the best uncertainty estimates.

10. Conclusions

In this article, we assessed and advanced the state of the art in predicting the performance of algorithms for hard combinatorial problems. We proposed new techniques for building predictive models, with a particular focus on improving prediction accuracy for parametric algorithms, and also introduced a wealth of new features for three of the most widely studied NP-hard problems (SAT, MIP and TSP) that benefit all models. We conducted the largest experimental study of which we are aware—predicting the performance of 11 algorithms on 35 instance distributions from SAT, MIP and TSP—comparing our new modeling approaches with a comprehensive set of methods from the literature. We showed that our new approaches—chiefly those based on random forests, but also approximate Gaussian processes—offer the best performance, whether we consider predictions for previously unseen problem instances for nonparametric algorithms, new parameter settings for a parametric algorithm running on a single problem instance, or parametric algorithms being run both with new parameter values and on previously unseen problem instances. We also demonstrated in each of

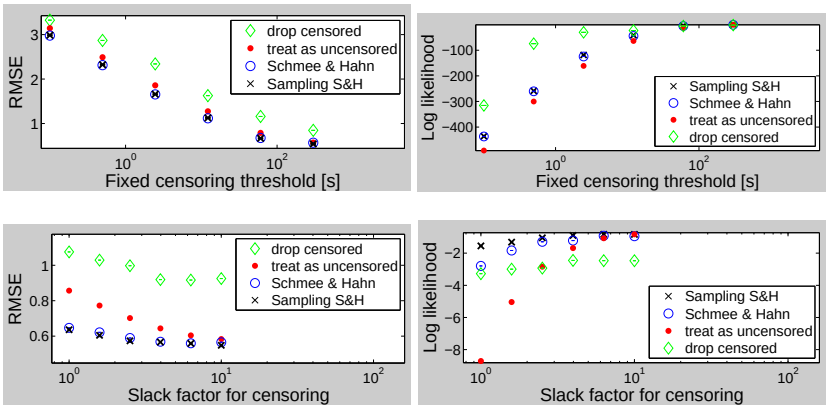


Figure 14: RMSE and log likelihood of four ways of handling censored data with random forests, for various levels of aggressiveness in setting the censoring threshold. Top: fixed thresholds; bottom: instance-specific thresholds. In both cases, larger numbers mean less censoring. Benchmark: CPLEX-BIGMIX. Results for all benchmarks are given in Figures B.40–B.48 (in the online appendix).

these settings that very accurate predictions (correlation coefficients between predicted and true runtime exceeding 0.9) are possible based on very small amounts of training data (only hundreds of runtime observations). Finally, we demonstrated how our best-performing model, random forests, could be improved further by better handling right-censored data stemming from unsuccessful runs that had been terminated prematurely. Overall, we showed that our methods are fast, general, and achieve good, robust performance; we hope they will be useful to a wide variety of researchers who seek to model algorithm performance for algorithm analysis, scheduling, algorithm portfolio construction, automated algorithm configuration, and other applications. The Matlab source code for our models, the data and source code to reproduce our experiments, and an online appendix containing additional experimental results, are available online at http://www.cs.ubc.ca/labs/beta/Projects/EPMs.

Acknowledgments

We thank Kevin Murphy for many valuable discussions regarding Gaussian processes and random forests. Thanks also to Jonathan Shen for proofreading an early version of this paper.

Appendix A. Details on Benchmark Instance Sets

We now briefly describe our instance benchmarks. For the SAT benchmarks, the number of variables and clauses are given for the original instance (before preprocessing).[11]

[11] In contrast, [44] reported these numbers after preprocessing, explaining the differences for IBM and SWV.

Appendix A.1. SAT benchmarks

INDU. This benchmark data set comprises 1 676 instances from the industrial categories of the 2002–2009 SAT competitions as well as from the 2006, 2008 and 2010 SAT Races. These instances contain an average of 111 000 variables and 689 187 clauses, with respective standard deviations of 318 955 and 1 510 764, and respective maxima of 9 685 434 variables and 14 586 886 clauses.

HAND. This benchmark data set comprises 1 955 instances from the handmade categories of the 2002-2009 SAT Competitions. These instances contain an average of 4 968 variables and 82 594 clauses, with respective standard deviations of 21 312 and 337 760, and respective maxima of 270 000 variables and 4 333 038 clauses.

RAND. This benchmark data set comprises 3 381 instances from the random categories of the 2002-2009 SAT Competitions. These instances contain an average of 1 048 variables and 6 626 clauses, with respective standard deviations of 2 593 and 11 221, and respective maxima of 19 000 variables and 79, 800 clauses.

COMPETITON. This set is the union of INDU, HAND, and RAND.

IBM. This set of SAT-encoded bounded model checking instances comprises 765 instances generated by Zarpas [112]; these instances were selected as the instances in 40 randomly-selected folders from the IBM Formal Verification Benchmarks Library. These instances contained an average of 96 454 variables and 413 143 clauses, with respective standard deviations of 169 859 and 717 379, and respective maxima of 1 621 756 variables and 6 359 302 clauses.

SWV. This set of SAT-encoded software verification instances comprises 604 instances generated with the CALYSTO static checker [4], used for the verification of five programs: the spam filter Dspam, the SAT solver HyperSAT, the Wine Windows OS emulator, the gzip archiver, and a component of xinetd (a secure version of inetd). These instances contain an average of 68 935 variables and 206 147 clauses, with respective standard deviations of 56 966 and 181 714, and respective maxima of 280 972 variables and 926 872 clauses.

RANDSAT. This set contains 2 076 satisfiable instances (proved by at least one winning solver from the previous SAT competitions) from data set RAND. These instances contain an average of 1 380 variables and 8 042 clauses, with respective standard deviations of 3 164 and 13 434, and respective maxima of 19 000 variables and 79, 800 clauses.

Appendix A.2. MIP benchmarks

BIGMIX. This highly heterogenous mix of publicly available Mixed Integer Linear Programming (MILP) benchmarks comprises 1 510 MILP instances. The instances in this set have an average of 8 610 variables and 4 250 constraints, with respective standard deviations of 34 832 and 21 009, and respective maxima of 550 539 variables and 550 339 constraints.

CORLAT. This set comprises 2 000 MILP instances based on real data used for the construction of a wildlife corridor for grizzly bears in the Northern Rockies region (the instances were described by Gomes et al. [30] and made available to us by Bistra Dilkina). All instances had 466 variables; on average they had 486 constraints (with standard deviation 25.2 and a maximum of 551).

RCW. This set comprises 1 980 MILP instances from a computational sustainability project. These instances model the spread of the endangered red-cockaded woodpecker, conditional on decisions about certain parcels of land to be protected. We generated 1 980 instances (20 random instances for each combination of 9 maps and 11 budgets), using the generator from [1] with the same parameter setting, except a smaller sample size of 5. All instances have 82 346 variables; on average, they have 328 816 constraints (with a standard deviation of only 3 and a maximum of 328 820).

REG. This set comprises 2 000 MILP-encoded instances of the winner determination problem in combinatorial auctions. We generated 2 000 instances using the `regions` generator from the Combinatorial Auction Test Suite [70], with the number of bids selected uniformly at random from between 750 and 1250, and a fixed bids/goods ratio of 3.91 (following [69]). They have an average of 1 129 variables and 498 constraints, with respective standard deviations of 73 and 32 and respective maxima of 1 255 variables and 557 constraints.

Appendix A.3. TSP benchmarks

RUE. This set comprises 4 993 uniform random Euclidean 2-dimensional TSP instances generated by the random TSP generator, `portgen` [57]. The number of nodes was randomly selected from 100 to 1 600, and the generated TSP instances contain an average of 849 nodes with a standard deviation of 429 and a maximum of 1 599 nodes.

RCE. This set comprises 5 001 random clustered Euclidean 2-dimensional TSP instances generated by the random TSP generator, portcgen [57]. The number of nodes was randomly selected from 100 to 1 600, and the number of clusters was set to 1% of the number of nodes. The generated TSP instances contain an average of 852 nodes with a standard deviation of 432 and a maximum of 1 599 nodes.

TSPLIB. This set contains a subset of the prominent TSPLIB (http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/) repository. We only included the 63 instances for which both our own feature computation code and the code by Smith-Miles & van Hemert [98] completed successfully (ours succeeded on 23 additional instances). These 63 instances have 931 ± 1376 nodes, with a range from 100 to 5 934.

References

[1] Ahmadizadeh, K., Dilkina, B.and Gomes, C., & Sabharwal, A. (2010). An empirical study of optimization for maximizing diffusion in networks. In *Proceedings of the 16th International Conference on Principles and Practice of Constraint Programming (CP'10)*, volume 6308 of *LNCS*, (pp. 514–521). Springer-Verlag.

[2] Applegate, D. L., Bixby, R. E., Chvátal, V., & Cook, W. J. (2006). *The traveling salesman problem: a computational study*. Princeton University Press.

[3] Babić, D. (2008). *Exploiting structure for scalable software verification*. PhD thesis, University of British Columbia, Vancouver, Canada.

[4] Babić, D. & Hu, A. J. (2007). Structural abstraction of software verification conditions. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07)*, volume 4590 of *LNCS*, (pp. 366–378). Springer-Verlag.

[5] Babić, D. & Hutter, F. (2007). Spear theorem prover. Solver description, SAT competition 2007.

[6] Bartz-Beielstein, T. (2006). *Experimental research in evolutionary computation: the new experimentalism*. Natural Computing Series. Springer.

[7] Bartz-Beielstein, T., Lasarczyk, C., & Preuss, M. (2005). Sequential parameter optimization. In *Proceedings of the 2004 Congress on Evolutionary Computation (CEC'05)*, (pp. 773–780).

[8] Bartz-Beielstein, T. & Markon, S. (2004). Tuning search algorithms for real-world applications: a regression tree based approach. In *Proceedings of the 2004 Congress on Evolutionary Computation (CEC'04)*, (pp. 1111–1118).

[9] Berkelaar, M., Dirks, J., Eikland, K., Notebaert, P., & Ebert, J. (2012). lp_solve 5.5. http://lpsolve.sourceforge.net/5.5/index.htm. Last accessed on August 6, 2012.

[10] Berthold, T., Gamrath, G., Heinz, S., Pfetsch, M., Vigerske, S., & Wolter, K. (2012). SCIP 1.2.1.4. http://scip.zib.de/doc/html/index.shtml. Last accessed on August 6, 2012.

[11] Bishop, C. M. (2006). *Pattern recognition and machine learning*. Springer.

[12] Box, G. E. P. & Draper, N. (2007). *Response surfaces, mixtures, and ridge analyses (second edition)*. Wiley.

[13] Box, G. E. P. & Wilson, K. (1951). On the experimental attainment of optimum conditions (with discussion). *Journal of the Royal Statistical Society Series B*, *13*(1), 1–45.

[14] Breiman, L. (2001). Random forests. *Machine Learning*, *45*(1), 5–32.

[15] Breiman, L., Friedman, J. H., Olshen, R., & Stone, C. J. (1984). *Classification and regression trees*. Wadsworth.

[16] Brewer, E. A. (1994). *Portable high-performance supercomputing: high-level platform-dependent optimization*. PhD thesis, Massachusetts Institute of Technology.

[17] Brewer, E. A. (1995). High-level optimization via automated statistical modeling. In *Proceedings of the 5th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPOPP-95)*, (pp. 80–91).

[18] Cheeseman, P., Kanefsky, B., & Taylor, W. M. (1991). Where the really hard problems are. In *Proceedings of the 9th National Conference on Artificial Intelligence (AAAI'91)*, (pp. 331–337).

[19] Chiarandini, M. & Goegebeur, Y. (2010). Mixed models for the analysis of optimization algorithms. In T. Bartz-Beielstein, M. Chiarandini, L. Paquete, & M. Preuss (Eds.), *Experimental Methods for the Analysis of Optimization Algorithms* (pp. 225–264). Springer-Verlag.

[20] Cook, W. (2012a). Applications of the TSP. http://www.tsp.gatech.edu/apps/index.html. Last accessed on April 10, 2012.

[21] Cook, W. (2012b). Concorde downloads page. http://www.tsp.gatech.edu/concorde/downloads/downloads.htm. Last accessed on October 24, 2012.

[22] Eén, N. & Biere, A. (2005). Effective preprocessing in SAT through variable and clause elimination. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, volume 3569 of *LNCS*, (pp. 61–75). Springer-Verlag.

[23] Eén, N. & Sörensson, N. (2004). An extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, (pp. 502–518).

[24] Ertin, E. (2007). Gaussian process models for censored sensor readings. In *Proceedings of the IEEE Statistical Signal Processing Workshop 2007 (SSP'07)*, (pp. 665–669).

[25] Gagliolo, M. & Legrand, C. (2010). Algorithm survival analysis. In T. Bartz-Beielstein, M. Chiarandini, L. Paquete, & M. Preuss (Eds.), *Experimental Methods for the Analysis of Optimization Algorithms* (pp. 161–184). Springer.

[26] Gagliolo, M. & Schmidhuber, J. (2006). Dynamic algorithm portfolios. In *International Symposium on Artificial Intelligence and Mathematics (ISAIM'06)*.

[27] Gebruers, C., Guerri, A., Hnich, B., & Milano, M. (2004). Making choices using structure at the instance level within a case based reasoning framework. In *Proceedings of the International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'04)*, volume 3011 of *LNCS*, (pp. 380–386). Springer-Verlag.

[28] Gebruers, C., Hnich, B., Bridge, D., & Freuder, E. (2005). Using CBR to select solution strategies in constraint programming. In *Proceedings of the 6th International Conference on Case Based Reasoning (ICCBR'05)*, volume 3620 of *LNCS*, (pp. 222–236). Springer-Verlag.

[29] Gomes, C. P., Selman, B., Crato, N., & Kautz, H. (2000). Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, *24*(1), 67–100.

[30] Gomes, C. P., van Hoeve, W.-J., & Sabharwal, A. (2008). Connections in networks: a hybrid approach. In *Proceedings of the International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'08)*, volume 5015 of *LNCS*, (pp. 303–307). Springer-Verlag.

[31] Guerri, A. & Milano, M. (2004). Learning techniques for automatic algorithm portfolio selection. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI'04)*, (pp. 475–479).

[32] Guo, H. & Hsu, W. H. (2004). A learning-based algorithm selection meta-reasoner for the real-time

MPE problem. In *Proceedings of the 17th Australian Conference on Artificial Intelligence (AI'04)*, volume 3339 of *LNCS*, (pp. 307–318). Springer-Verlag.

[33] Gurobi Optimization Inc. (2012). Gurobi 2.0. http://www.gurobi.com/. Last accessed on August 6, 2012.

[34] Guyon, I., Gunn, S., Nikravesh, M., & Zadeh, L. (2006). *Feature extraction, foundations and applications*. Springer.

[35] Haim, S. & Walsh, T. (2008). Online estimation of SAT solving runtime. In *Proceedings of the 11th International Conference on Theory and Applications of Satisfiability Testing (SAT'08)*, volume 4996 of *LNCS*, (pp. 133–138). Springer-Verlag.

[36] Hastie, T., Tibshirani, R., & Friedman, J. H. (2009). *The elements of statistical learning (second edition)*. Springer Series in Statistics. Springer.

[37] Helsgaun, K. (2000). An effective implementation of the Lin-Kernighan traveling salesman heuristic. *European Journal of Operational Research*, *126*(1), 106–130.

[38] Herwig, P. (2006). Using graphs to get a better insight into satisfiability problems. Master's thesis, Delft University of Technology, Department of Electrical Engineering, Mathematics and Computer Science.

[39] Hoos, H. H. & Stützle, T. (2005). *Stochastic local search – foundations & applications*. Morgan Kaufmann Publishers.

[40] Horvitz, E., Ruan, Y., Gomes, C. P., Kautz, H., Selman, B., & Chickering, D. M. (2001). A Bayesian approach to tackling hard computational problems. In *Proceedings of the 17th Conference on Uncertainty in Artificial Intelligence (UAI'01)*, (pp. 235–244).

[41] Hothorn, T., Lausen, B., Benner, A., & Radespiel-Tröger, M. (2004). Bagging survival trees. *Statistics in Medicine*, *23*, 77–91.

[42] Hsu, E. I., Muise, C., Beck, J. C., & McIlraith, S. A. (2008). Probabilistically estimating backbones and variable bias: experimental overview. In *Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming (CP'08)*, volume 5202 of *LNCS*, (pp. 613–617). Springer-Verlag.

[43] Huang, L., Jia, J., Yu, B., Chun, B., P.Maniatis, & Naik, M. (2010). Predicting execution time of computer programs using sparse polynomial regression. In *Proceedings of the 23rd Conference on Advances in Neural Information Processing Systems (NIPS'10)*, (pp. 883–891).

[44] Hutter, F. (2009). *Automated configuration of algorithms for solving hard computational problems*. PhD thesis, University Of British Columbia, Department of Computer Science, Vancouver, Canada.

[45] Hutter, F., Babić, D., Hoos, H. H., & Hu, A. J. (2007). Boosting verification by automatic tuning of decision procedures. In *Proceedings of the 7th International Conference on Formal Methods in Computer-Aided Design (FMCAD'07)*, (pp. 27–34).

[46] Hutter, F., Hamadi, Y., Hoos, H. H., & Leyton-Brown, K. (2006). Performance prediction and automated tuning of randomized and parametric algorithms. In *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming (CP'06)*, volume 4204 of *LNCS*, (pp. 213–228). Springer-Verlag.

[47] Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2010a). Automated configuration of mixed integer programming solvers. In *Proceedings of the International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'10)*, volume 6140 of *LNCS*, (pp. 186–202). Springer-Verlag.

[48] Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2010b). Tradeoffs in the empirical evaluation of competing algorithm designs. *Annals of Mathematics and Artificial Intelligenc (AMAI), Special Issue on Learning and Intelligent Optimization*, *60*(1), 65–89.

[49] Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2011a). Bayesian optimization with censored response data. In *NIPS 2011 workshop on Bayesian Optimization, Sequential Experimental Design, and Bandits*. Published online.

[50] Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2011b). Sequential model-based optimization for general algorithm configuration. In *Proceedings of the 5th Workshop on Learning and Intelligent Optimization (LION'11)*, volume 6683 of *LNCS*, (pp. 507–523). Springer-Verlag.

[51] Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2012). Parallel algorithm configuration. In *Proceedings*

of the 6th Workshop on Learning and Intelligent Optimization (LION'12), LNCS, (pp. 55–70). Springer-Verlag.

[52] Hutter, F., Hoos, H. H., Leyton-Brown, K., & Murphy, K. P. (2009). An experimental investigation of model-based parameter optimisation: SPO and beyond. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'09)*, (pp. 271–278).

[53] Hutter, F., Hoos, H. H., Leyton-Brown, K., & Murphy, K. P. (2010). Time-bounded sequential parameter optimization. In *Proceedings of the 4th Workshop on Learning and Intelligent Optimization (LION'10)*, (pp. 281–298).

[54] Hutter, F., Tompkins, D. A. D., & Hoos, H. H. (2002). Scaling and probabilistic smoothing: efficient dynamic local search for SAT. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP'02)*, volume 2470 of *LNCS*, (pp. 233–248). Springer-Verlag.

[55] International Business Machines Corp. (2011). IBM ILOG CPLEX Optimizer – Data Sheet. ftp://public.dhe.ibm.com/common/ssi/ecm/en/wsd14044usen/WSD14044USEN.PDF. Last accessed on August 6, 2012.

[56] International Business Machines Corp. (2012). CPLEX 12.1. http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/. Last accessed on August 6, 2012.

[57] Johnson, D. S. (2011). Random TSP generators for the DIMACS TSP challenge. http://www2.research.att.com/~dsj/chtsp/codes.tar. Last accessed on May 16, 2011.

[58] Jones, D. R., Perttunen, C. D., & Stuckman, B. E. (1993). Lipschitzian optimization without the Lipschitz constant. *Journal of Optimization Theory and Applications*, *79*(1), 157–181.

[59] Jones, D. R., Schonlau, M., & Welch, W. J. (1998). Efficient global optimization of expensive black box functions. *Journal of Global Optimization*, *13*, 455–492.

[60] Jones, T. & Forrest, S. (1995). Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In *Proceedings of the 6th International Conference on Genetic Algorithms (ICGA'95)*, (pp. 184–192).

[61] Kadioglu, S., Malitsky, Y., Sellmann, M., & Tierney, K. (2010). ISAC - instance specific algorithm configuration. In *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI'10)*, (pp. 751–756).

[62] Kilby, P., Slaney, J., Thiebaux, S., & Walsh, T. (2006). Estimating search tree size. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI'06)*, (pp. 1014–1019).

[63] Knuth, D. (1975). Estimating the efficiency of backtrack programs. *Mathematics of Computation*, *29*(129), 121–136.

[64] Krige, D. G. (1951). A statistical approach to some basic mine valuation problems on the Witwatersrand. *Journal of the Chemical, Metallurgical and Mining Society of South Africa*, *52*(6), 119–139.

[65] Lawrence, N. D., Seeger, M., & Herbrich, R. (2003). Fast sparse Gaussian process methods: the informative vector machine. In *Proceedings of the 15th Conference on Advances in Neural Information Processing Systems (NIPS'02)*, (pp. 609–616).

[66] Leyton-Brown, K., Nudelman, E., Andrew, G., McFadden, J., & Shoham, Y. (2003a). Boosting as a metaphor for algorithm design. In *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming (CP'03)*, volume 2833 of *LNCS*, (pp. 899–903). Springer-Verlag.

[67] Leyton-Brown, K., Nudelman, E., Andrew, G., McFadden, J., & Shoham, Y. (2003b). A portfolio approach to algorithm selection. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI'03)*, (pp. 1542–1543).

[68] Leyton-Brown, K., Nudelman, E., & Shoham, Y. (2002). Learning the empirical hardness of optimization problems: the case of combinatorial auctions. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP'02)*, volume 2470 of *LNCS*, (pp. 556–572). Springer-Verlag.

[69] Leyton-Brown, K., Nudelman, E., & Shoham, Y. (2009). Empirical hardness models: methodology and a case study on combinatorial auctions. *Journal of the ACM*, *56*(4), 1–52.

[70] Leyton-Brown, K., Pearson, M., & Shoham, Y. (2000). Towards a universal test suite for combinatorial

auction algorithms. In *EC '00: Proceedings of the 2nd ACM conference on Electronic commerce*, (pp. 66–76).

[71] Lin, S. & Kernighan, B. W. (1973). An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, *21*(2), 498–516.

[72] Lobjois, L. & Lemaître, M. (1998). Branch and bound algorithm selection by performance prediction. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98)*, (pp. 353–358).

[73] Mahajan, Y. S., Fu, Z., & Malik, S. (2005). Zchaff2004: an efficient SAT solver. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, volume 3542 of *LNCS*, (pp. 360–375). Springer-Verlag.

[74] Meinshausen, N. (2006). Quantile regression forests. *Journal of Machine Learning Research*, *7*, 983–999.

[75] Mitchell, D., Selman, B., & Levesque, H. (1992). Hard and easy distributions of SAT problems. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI'92)*, (pp. 459–465).

[76] Nabney, I. T. (2002). *NETLAB: algorithms for pattern recognition*. Springer.

[77] Nelson, W. (2003). *Applied Life Data Analysis*. Wiley Series in Probability and Statistics. John Wiley & Sons.

[78] Nocedal, J. & Wright, S. J. (2006). *Numerical optimization (second edition)*. Springer.

[79] Nudelman, E., Leyton-Brown, K., Hoos, H. H., Devkar, A., & Shoham, Y. (2004). Understanding random SAT: beyond the clauses-to-variables ratio. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming (CP'04)*, volume 3258 of *LNCS*, (pp. 438–452). Springer-Verlag.

[80] Pfahringer, B., Bensusan, H., & Giraud-Carrier, C. (2000). Meta-learning by landmarking various learning algorithms. In *Proceedings of the 17th International Conference on Machine Learning (ICML'00)*, (pp. 743–750).

[81] Prasad, M. R., Biere, A., & Gupta, A. (2005). A survey of recent advances in SAT-based formal verification. *International Journal on Software Tools for Technology Transfer*, *7*(2), 156–173.

[82] Quinonero-Candela, J., Rasmussen, C. E., & Williams, C. K. (2007). Approximation methods for Gaussian process regression. In *Large-Scale Kernel Machines*, Neural Information Processing (pp. 203–223). MIT Press.

[83] Rasmussen, C. E. & Williams, C. K. I. (2006). *Gaussian processes for machine learning*. MIT Press.

[84] Rice, J. R. (1976). The algorithm selection problem. *Advances in Computers*, *15*, 65–118.

[85] Ridge, E. & Kudenko, D. (2007). Tuning the performance of the MMAS heuristic. In *Proceedings of the International Workshop on Engineering Stochastic Local Search Algorithms (SLS'2007)*, volume 4638 of *LNCS*, (pp. 46–60). Springer-Verlag.

[86] Sacks, J., Welch, W. J., Welch, T. J., & Wynn, H. P. (1989). Design and analysis of computer experiments. *Statistical Science*, *4*(4), 409–423.

[87] Sander, J., Ester, M., Kriegel, H., & Xu, X. (1998). Density-based clustering in spatial databases: The algorithm GDBSCAN and its applications. *Data Mining and Knowledge Discovery*, *2*, 169–194.

[88] Santner, T. J., Williams, B. J., & Notz, W. I. (2003). *The design and analysis of computer experiments*. Springer.

[89] Schmee, J. & Hahn, G. J. (1979). A simple method for regression analysis with censored data. *Technometrics*, *21*(4), 417–432.

[90] Schmidt, M. (2012). minfunc. http://www.di.ens.fr/~mschmidt/Software/minFunc.html. Last accessed on August 5, 2012.

[91] Segal, M. R. (1988). Regression trees for censored data. *Biometrics*, *44*(1), 35–47.

[92] Shawe-Taylor, J. & Cristianini, N. (2004). *Kernel methods for pattern analysis*. Cambridge University Press.

[93] Sherman, J. & Morrison, W. J. (1949). Adjustment of an inverse matrix corresponding to changes in

the elements of a given column or a given row of the original matrix (abstract). *Annals of Mathematical Statistics*, *20*, 621.

[94] Smith-Miles, K. (2009). Cross-disciplinary perspectives on meta-learning for algorithm selection. *ACM Computing Surveys*, *41*(1), 6:1–6:25.

[95] Smith-Miles, K. & Lopes, L. (2011). Generalising algorithm performance in instance space: A timetabling case study. In *Proceedings of the 5th Workshop on Learning and Intelligent Optimization (LION'11)*, volume 6683 of *LNCS*, (pp. 524–539). Springer-Verlag.

[96] Smith-Miles, K. & Lopes, L. (2012). Measuring instance difficulty for combinatorial optimization problems. *Computers and Operations Research*, *39*(5), 875–889.

[97] Smith-Miles, K. & Tan, T. (2012). Measuring algorithm footprints in instance space. In *Proceedings of the 2012 Congress on Evolutionary Computation (CEC'12)*, (pp. 3446–3453).

[98] Smith-Miles, K. & van Hemert, J. (2011). Discovering the suitability of optimisation algorithms by learning from evolved instances. *Annals of Mathematics and Artificial Intelligence*, *61*, 87–104.

[99] Smith-Miles, K., van Hemert, J., & Lim, X. Y. (2010). Understanding TSP difficulty by learning from evolved instances. In *Proceedings of the 4th Workshop on Learning and Intelligent Optimization (LION'10)*, volume 6073 of *LNCS*, (pp. 266–280). Springer-Verlag.

[100] Smith-Miles, K., Wreford, B., Lopes, L., & Insani, N. (2012). In *Hybrid Metaheuristics*, Springer Series in Computational Intelligence, chapter Predicting metaheuristic performance on graph coloring problems using data mining. Springer. In press.

[101] Soos, M. (2010). CryptoMiniSat 2.5.0. Solver description, SAT Race 2010.

[102] Tresp, V. (2000). A Bayesian committee machine. *Neural Computation*, *12*(11), 2719–2741.

[103] Wei, W. & Li, C. M. (2009). Switching between two adaptive noise mechanisms in local search for SAT. Solver description, SAT competition 2009.

[104] Weinberger, E. (1990). Correlated and uncorrelated fitness landscapes and how to tell the difference. *Biological Cybernetics*, *63*, 325–336.

[105] Weiss, N. A. (2005). *A course in probability*. Addison–Wesley.

[106] Xu, L., Hoos, H. H., & Leyton-Brown, K. (2007). Hierarchical hardness models for SAT. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP'07)*, volume 4741 of *LNCS*, (pp. 696–711). Springer-Verlag.

[107] Xu, L., Hoos, H. H., & Leyton-Brown, K. (2010). Hydra: automatically configuring algorithms for portfolio-based selection. In *Proceedings of the 25th National Conference on Artificial Intelligence (AAAI'10)*, (pp. 210–216).

[108] Xu, L., Hutter, F., Hoos, H., & Leyton-Brown, K. (2007). SATzilla-07: the design and analysis of an algorithm portfolio for SAT. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP'07)*, volume 4741 of *LNCS*, (pp. 712–727). Springer-Verlag.

[109] Xu, L., Hutter, F., Hoos, H., & Leyton-Brown, K. (2009). SATzilla2009: an automatic algorithm portfolio for sat. Solver description, SAT competition 2009.

[110] Xu, L., Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2008). SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, *32*, 565–606.

[111] Xu, L., Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2012). Evaluating component solver contributions in portfolio-based algorithm selectors. In *Proceedings of the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT'12)*, volume 7317 of *LNCS*, (pp. 228–241). Springer-Verlag.

[112] Zarpas, E. (2005). Benchmarking SAT solvers for bounded model checking. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, volume 3569 of *LNCS*, (pp. 340–354). Springer-Verlag.